



Multithreading

An Extensive Study on Linux, OpenSolaris, and Windows

Dalmo Cirne

dalmo.cirne@gmail.com

<http://dalmocirne.blogspot.com>

Introduction	2
The Experiment	3
Hardware Configuration and Software Versions	6
Installing Linux, OpenSolaris, and Windows on the same machine	7
Installing the Development Tools on each O.S.	7
Linux	7
OpenSolaris	8
Windows	9
Obtaining the Source Code	9
Compiling the Programs and Running the Experiments	9
Methodology for Analyzing the Data	12
Analysis of the Data	13
Linear - C	13
Linear - Java	19
Threads - C	24
Threads - Java	28
Stress - C	34
Stress - Java	38
The Economics of Operating Systems	42
Conclusion	44

Abstract

There are many reasons why one may choose a particular operating system to run on their computer: availability of software, technical skills, etc. Among those reasons one may choose the operating system with the best performance, or the one with the lowest cost per transaction. This paper examines three operating systems (Linux, OpenSolaris, and Windows) and their respective capabilities of taking advantage of the multiple cores present in modern processors by executing operations with integers, floating points and math, and I/O in a multithreaded way, achieving parallelism in the execution of those operations, and then tries to reason the measurements from the experiments in the economics of an operating system.

Introduction

Often times the choice of an operating system is related to some platform dependent tools one will be using (e.g., programming languages, spreadsheets, video editors), the services that will be running (e.g., database servers, application servers, Email servers), or availability of professionals with knowledge on a particular system. However, often times the tools, services and professionals are available on any chosen platform. Therefore, knowing better about the capabilities of an operating system may be a decisive factor in choosing the one that is best for your needs.

Modern computers are rapidly increasing the number of cores available per processor. Two, four, and even 12 cores per chip are current available configurations. Therefore, the ability of an operating system to take advantage of such processors and scale its performance proportionally to the number of available cores is of great importance. An operating system should be able to manage threads efficiently, assign multiple threads to multiple cores when available, and use time slicing to simulate simultaneous execution when all cores are busy and more threads are scheduled to be executed. Linux and OpenSolaris use the Posix threads (pthreads) model and Microsoft Windows uses the Win32/Win64 model.

The ideas and experiments discussed throughout this paper are conceptually simple, nonetheless, it comes with several difficulties to be implemented into practice. The experiment includes

installing Linux, OpenSolaris, and Windows on the same machine¹ and running a series of operations with integer numbers, floating point and math, and Input/Output. Together these operations roughly express all a computer can do, hence allowing us to infer that other software applications will behave similarly.

The Experiment

We want to compare the performance of the three operating systems. We have to make them execute identical tasks and compare execution times for each of the tasks. These tasks have to take the form of a program that will be compiled and executed on top of each of the operating systems.

In this experiment we chose to compare the performance in two different scenarios. In the first scenario, a program written in C, used specific O.S. APIs, but executed the exact same operations. For the second scenario, the program was written in Java. This way we have the very same source code, but compiled and executed on different operating systems. C and Java are the best choices to run this experiment. With the exception of Assembler, C is as fast as it gets when it comes to run a program. Java is widespread used among many industries and implemented for various platforms including the ones we are testing. Moreover, Java Virtual Machine (JVM) threads are native threads, rather than *green threads*, typically found in other interpreted languages. Which means to say that Java delegates to the operating system (kernel space) the task of managing multiple threads, rather than emulating the multithreading environment (user space) to make it operating system independent.

As in any experiment, it is important to have a point of reference to compare to when the results of the multithreaded computations are complete. The logical choice in this case is to create programs that will execute exactly the same tasks as the multithreaded ones, however the tasks will be executed serially. First the programs will execute the operations with integers, second with floating point and math, and last the I/O operations. The total running time of each iteration will be about the sum of the times to complete each individual task, whereas in the case of the multithreaded experiment the expected running time of each iteration will be the time it takes for the longest operation to finish since all the operations will be running in parallel in separate threads.

¹ The choice to install all three operating systems on the same machine removes environment errors due to exogenous factors.

The third and last experiment will consist of putting the operating system under a condition of considerable stress. There will be many more threads than the number of available cores and each of the operations are computationally intensive. As a result, it is expected that each operation will take much longer to finish. Moreover, the stress experiment will take longer to finish than the linear experiment. The rationale is the overhead introduced by requiring from the operating system the constant switching among the running threads and managing a very stressful environment, with simultaneous operations competing for a limited number of resources. It is important to run the experiment this way, because we want to quantify the ability of an operating system to manage the concurrency of multiple threads.

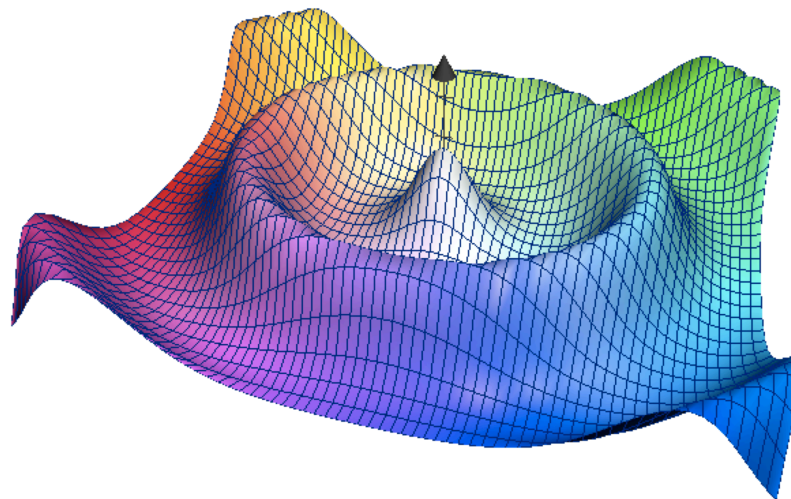
The operations with integers are the simplest to implement. A global variable called *counter* is assigned to a temp variable, then we add 37, subtract 36, and assign the result back to *counter*. Effectively we are just incrementing *counter* by 1 at each iteration.

$$\begin{aligned}temp &\leftarrow counter \\temp &\leftarrow temp + 37 \\temp &\leftarrow temp - 36 \\counter &\leftarrow temp\end{aligned}$$

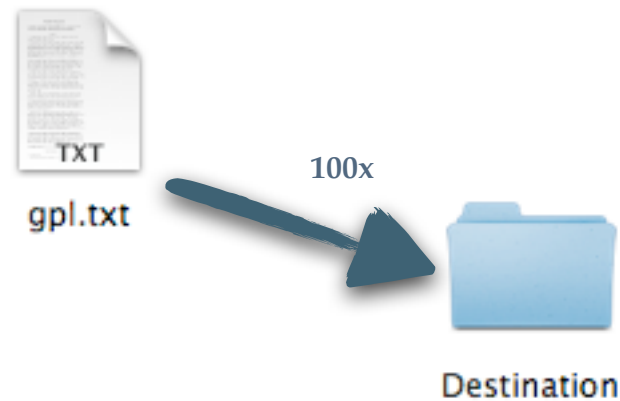
The operations with floating point and math are executed by solving the following equation:

$$\begin{bmatrix} z \\ x \\ y \end{bmatrix} = \begin{bmatrix} \exp(\cos(\sqrt{x^2 + y^2})) \\ i/1.1 \\ i \times 1.1 \end{bmatrix}$$

When solved for the intervals $x = [-10, 10]$ and $y = [-10, 10]$ and plotted, it renders the curve:



Lastly, the I/O operations consist of reading a file and replicating it a certain number of times. In this case we chose to read and replicate the GPL.



In the stress experiment we will execute the operations with integers, and floating point and math a little differently from its linear and multithreaded counterparts. One hundred concurrent threads will be competing to increment *counter*, however only one thread at a time will be able to increment it successfully. A mutex is used to manage the concurrency, delegating to the operating system the task of efficiently determining which thread wins the competition, and which threads wait idle for their turn to run. Eventually each of the 100 threads will be executed.

Another difference introduced in the stress experiment is the usage of the classical Producer and Consumer model for the operations with floating point and math. The consumer thread asks for the result of a calculation. If the result has not yet been published by the producer thread, the consumer thread goes into a wait state and sleeps until it receives a notification from the producer thread that the result is available to be consumed. After notifying the consumer thread, the producer thread will go into a sleep state and will wait until it receives a notification from the consumer thread that the previous result has been consumed and that it is ok to produce the next one.

Each of the experiments iterates 100 times per execution. This way we can generate data with a significant statistical mass on each one of the 100 iterations. We executed the integer operations 100,000,000 times; the floating point and math operations 200,000 times; and replicated the GPL 100 times².

² On each iteration we create a different directory to host the copies of the GPL. In the end of each run there will be 100 subdirectories, each one containing 100 copies of the GPL.

Hardware Configuration and Software Versions

A machine capable of running this experiment has to have certain minimum requirements. For example, it must have a processor with more than one core and enough RAM not to use swap memory. It needs more than one core because we want to see tasks being executed in parallel. And it needs enough RAM because we don't want to distort the I/O performance test by having the operating system swapping data between physical memory and disk. The following table contains the machine configuration used in this experiment:

	CONFIGURATION
PROCESSOR	Intel Core 2 Quad 2.5 GHz
RAM	2x2GB DDR2 1066 PC2-8500
VIDEO CARD	NVidia 8600GT 512 MB
HARD DISK	Western Digital WD5000AAKS - 500GB - 7200 RPM - SATA2
MAIN BOARD	ASUS P5K-E P35 775

The software configuration is not identical across platforms, but we tried to keep them as homogeneous as possible. The next table shows each operating system used in this experiment and the softwares used to compile and run the programs.

	LINUX	OPENSOLARIS	WINDOWS
O.S. VERSION	Fedora 12	2009.06 (20091230)	Windows 7 Professional
DISK PARTITION	/dev/sda3	/dev/sda2	/dev/sda1
JAVA	OpenJDK - 64bit 1.6.0_0 build 14.0_b16	SUN JDK - 64bit 1.6.0_17-b04	SUN JDK - 64bit 1.6.0_17-b04
C	gcc 4.4.2	gcc 4.4.2	Visual C++ 2008 Express Edition 9.0.30729.1SP
FILESYSTEM	ext4	ZFS	NTFS

Installing Linux, OpenSolaris, and Windows on the same machine

The first challenge of this experiment was to install and boot each operating system separately on the same machine. Linux and OpenSolaris actually didn't put up much of a challenge, most of the difficulties were related to getting Windows to behave as a good citizen and not assume that the entire machine was available to it.

The only way to get it all working was to install Windows first and the other two operating systems afterwards. In my case I ran the Windows installer, partitioned the disk in three equal partitions and used the first partition to install Windows³. The other two partitions were left untouched for Linux and OpenSolaris.

Next in-line was OpenSolaris. No surprises here. OpenSolaris was installed in the second disk partition, and the boot manager recognized the Windows partition and added it as a bootable option.

Last, but not least, Linux. The installation also ran with no surprises, taking the third and last partition on the disk. However during the boot manager configuration, it did not automatically recognize OpenSolaris. Nonetheless, since it offers an option to edit the boot options you simply need to add `"/dev/sda2"` and label it as `"OpenSolaris"`. When selected from the Linux boot manager we are taken to the OpenSolaris boot manager and from there all works fine.

Installing the Development Tools on each O.S.

Linux

The Fedora installer gives you the option to install OpenJDK and gcc. If you select those at install time, you're good to go. Otherwise you can install them easily from the command line

```
# yum install java-1.6.0-openjdk
# yum install gcc
```

³ No anti-virus software was installed with Windows to avoid a performance hit. However, it is highly recommended the installation of an anti-virus program in computers running Windows.

OpenSolaris

The OpenSolaris installer does not give you the option to install either Java or gcc. Those have to be installed later. Java has to be installed in two steps (if we want the 64-bit version). Another two steps will be necessary to install gcc version 4.4.2.

Detailed instructions on how to install Java on OpenSolaris can be found at:

<http://java.sun.com/javase/6/webnotes/install/jdk/install-solaris.html> - 32-bit version

<http://java.sun.com/javase/6/webnotes/install/jdk/install-solaris-64.html> - 64-bit version

Notice that it is necessary to install the 32-bit version of Java prior to installing the 64-bit one.

We will have to download gcc 4.4.2 source and compile it, but before we do that we need to have a C compiler installed. We can install gcc 3.4.3 using Sun's package manager. On a terminal type:

```
$ pfexec pkg install SUNWgcc
$ pfexec pkg install SUNWgnu-mp
$ pfexec pkg install SUNWgnu-mpfr
```

gcc's source code can be downloaded from any mirror listed in the GNU Compiler Collection site <http://gcc.gnu.org>. Once the source has been downloaded, change to the directory you downloaded it, uncompress, compile and install it.

```
$ tar xvjf gcc-4.4.2.tar.bz2
$ cd gcc-4.4.2
$ mkdir objdir/
$ cd objdir/
$ ../configure --prefix=/usr/local \
  --with-mpfr-include=/usr/include/mpfr/ \
  --with-as=/usr/sfw/bin/gas --with-gnu-as \
  --with-ld=/usr/ccs/bin/ld --without-gnu-ld \
  --enable-shared --with-gmp-include=/usr/include/gmp/
$ gmake
$ pfexec gmake install
```

Windows

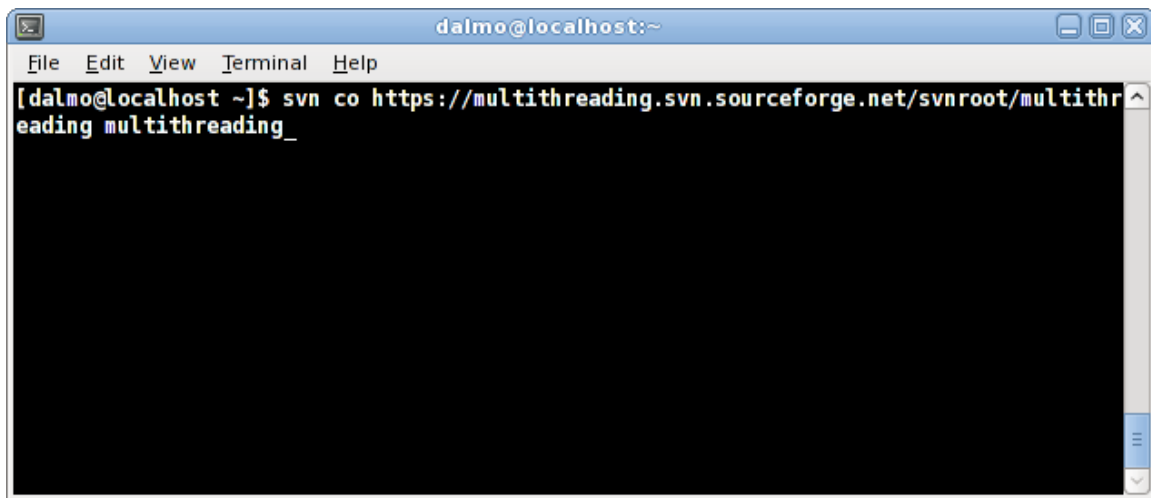
The Windows installer does not give you the option to install either Java or Visual C++ 2008 Express Edition. However installing both is quite simple. Just download the JDK from Sun's web site and Visual C++ from Microsoft's web site. Both are listed below:

<http://java.sun.com/javase/downloads/widget/jdk6.jsp>

<http://www.microsoft.com/express/Downloads/#2008-Visual-CPP>

Obtaining the Source Code

All the source code is hosted at SourceForge⁴. You can checkout the code using Subversion, open a shell and type:

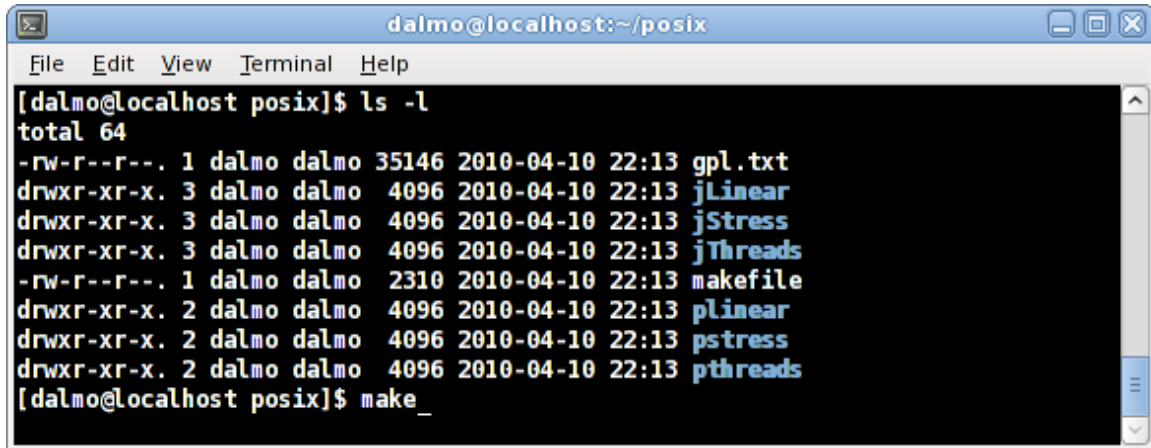
A screenshot of a terminal window titled 'dalmo@localhost:~'. The window has a menu bar with 'File', 'Edit', 'View', 'Terminal', and 'Help'. The terminal content shows a Subversion checkout command: '[dalmo@localhost ~]\$ svn co https://multithreading.svn.sourceforge.net/svnroot/multithreading multithreading_'. The command is partially visible on two lines: 'eading multithreading_' on the first line and 'eading multithreading_' on the second line. The terminal background is black with white text.

svn co <https://multithreading.svn.sourceforge.net/svnroot/multithreading> multithreading

Compiling the Programs and Running the Experiments

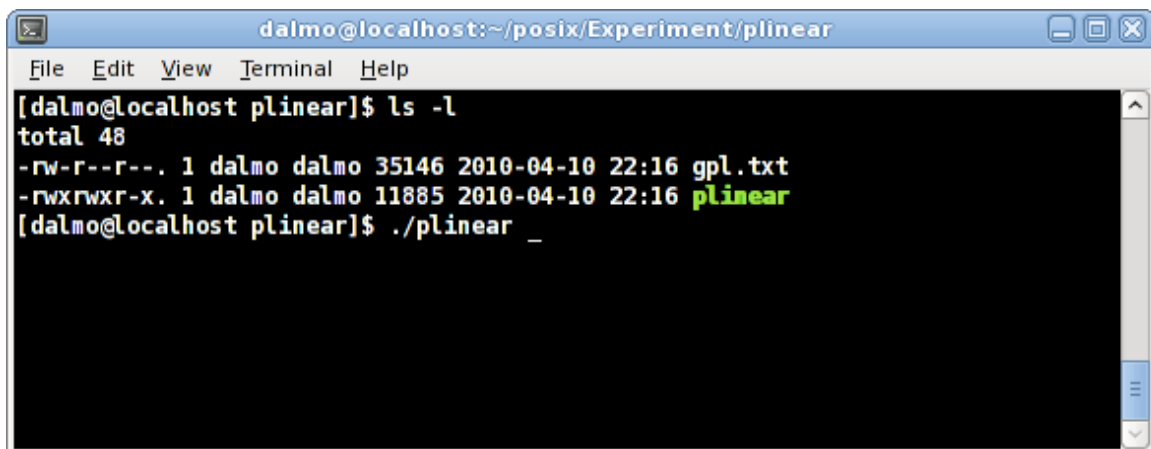
After you download the programs you will need to compile them before running the experiments. For Linux and OpenSolaris there is a convenience *makefile*, on the root directory where you put the source code that will compile and organize the binaries for you. If you open the file *makefile*, you will see all the options used to compile and link the programs. Notice, however, that you may need to inform the location of the *gcc* compiler. A suggestion is already written in the *makefile*. Open it and get familiarized with the compilation and linkage options. On the shell prompt change to the *posix* directory and type: *make*.

⁴ <http://multithreading.sourceforge.net>



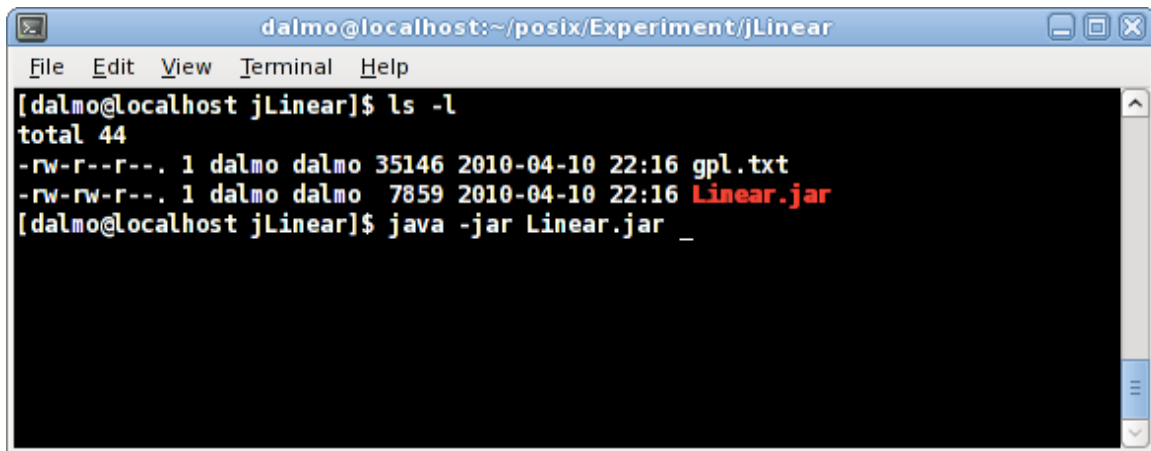
```
dalmo@localhost:~/posix
File Edit View Terminal Help
[dalmo@localhost posix]$ ls -l
total 64
-rw-r--r--. 1 dalmo dalmo 35146 2010-04-10 22:13 gpl.txt
drwxr-xr-x. 3 dalmo dalmo 4096 2010-04-10 22:13 jLinear
drwxr-xr-x. 3 dalmo dalmo 4096 2010-04-10 22:13 jStress
drwxr-xr-x. 3 dalmo dalmo 4096 2010-04-10 22:13 jThreads
-rw-r--r--. 1 dalmo dalmo 2310 2010-04-10 22:13 makefile
drwxr-xr-x. 2 dalmo dalmo 4096 2010-04-10 22:13 plinear
drwxr-xr-x. 2 dalmo dalmo 4096 2010-04-10 22:13 pstress
drwxr-xr-x. 2 dalmo dalmo 4096 2010-04-10 22:13 pthreads
[dalmo@localhost posix]$ make _
```

All files pertinent to the experiments are going to be inside of a newly created directory called *Experiment*. Inside of the *Experiment* directory there will be a directory for each individual experiment. To run the experiments using C compiled binaries, just change to the experiments directory and execute the binary, for example, the linear experiment:



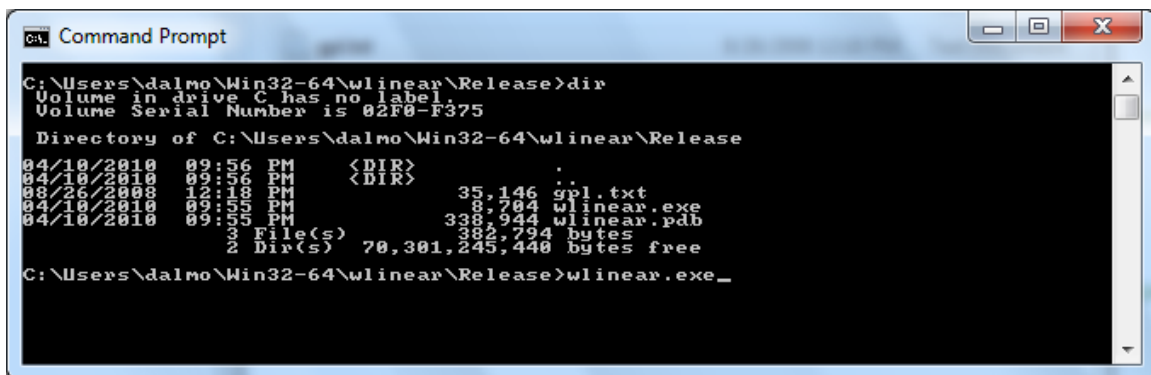
```
dalmo@localhost:~/posix/Experiment/plinear
File Edit View Terminal Help
[dalmo@localhost plinear]$ ls -l
total 48
-rw-r--r--. 1 dalmo dalmo 35146 2010-04-10 22:16 gpl.txt
-rwxrwxr-x. 1 dalmo dalmo 11885 2010-04-10 22:16 plinear
[dalmo@localhost plinear]$ ./plinear _
```

The experiments using Java will also be ran from the shell prompt. For example, the linear experiment:



```
dalmo@localhost:~/posix/Experiment/jLinear
File Edit View Terminal Help
[dalmo@localhost jLinear]$ ls -l
total 44
-rw-r--r--. 1 dalmo dalmo 35146 2010-04-10 22:16 gpl.txt
-rw-rw-r--. 1 dalmo dalmo 7859 2010-04-10 22:16 Linear.jar
[dalmo@localhost jLinear]$ java -jar Linear.jar _
```

For the Windows case the you will need to open the projects of the programs written in C on Visual C++ and build each of them. Open the project, select *Release* from the Solution Configuration drop-down box, and *Build Solution* (F7) the project. You will find the binary inside the *Release* directory under your project's root directory. The last step is to copy the file "gpl.txt" to that directory and run the experiment. For example:

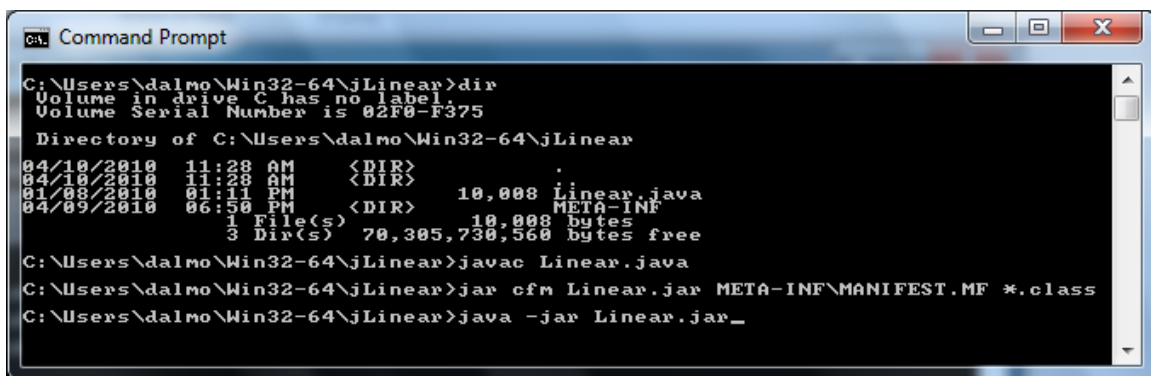


```
ca. Command Prompt
C:\Users\dalmo\Win32-64\wlinear\Release>dir
Volume in drive C has no label.
Volume Serial Number is 02F0-F375

Directory of C:\Users\dalmo\Win32-64\wlinear\Release
04/10/2010 09:56 PM <DIR> .
04/10/2010 09:56 PM <DIR> ..
08/26/2008 12:18 PM 35,146 gpl.txt
04/10/2010 09:56 PM 338,704 wlinear.exe
04/10/2010 09:56 PM 338,944 wlinear.pdb
no File(s)
no Dir(s) 70,301,245,440 bytes free

C:\Users\dalmo\Win32-64\wlinear\Release>wlinear.exe_
```

Also for Windows we will need to compile the Java based programs. Change to the directory of the source .java file, compile the source code and pack the bytecode using jar. Then copy the file "gpl.txt" to that directory and run the experiment. For example, the linear experiment in Java:



```
ca. Command Prompt
C:\Users\dalmo\Win32-64\jLinear>dir
Volume in drive C has no label.
Volume Serial Number is 02F0-F375

Directory of C:\Users\dalmo\Win32-64\jLinear
04/10/2010 11:28 AM <DIR> .
04/10/2010 11:28 AM <DIR> ..
01/08/2010 01:11 PM 10,008 Linear.java
04/09/2010 06:50 PM <DIR> META-INF
1 File(s) 10,008 bytes
3 Dir(s) 70,305,730,560 bytes free

C:\Users\dalmo\Win32-64\jLinear>javac Linear.java
C:\Users\dalmo\Win32-64\jLinear>jar cfm Linear.jar META-INF\MANIFEST.MF *.class
C:\Users\dalmo\Win32-64\jLinear>java -jar Linear.jar _
```

All experiments output their respective collected results in a .csv file, named hinting the operating system, the experiment, and whether it was executed from a C compiled program or Java. There you can find the sample data that is necessary to run the analysis contained in this paper.

Methodology for Analyzing the Data

The data collected from these experiments is a very large sample of the time it took for each operation (integers, floating point and math, and I/O) to complete, plus the time each iteration took to complete (100 iterations were executed per program in the experiments). For each operating system, a set of programs were executed to measure the overall performance to complete the operations. One program in C executing the linear operations, one program in C executing the multithreaded operations, and one program in C executing the stressed operations. As well as one program in Java executing the linear operations, one program in Java executing the multithreaded operations, and one program in Java executing the stressed operations. In total there were 18 experiments ran (3 operating systems x 6 experiments on each operating system).

The sample data allows for the computation, with great statistical significance, of several statistics:

- *Average Running Time*: This will give us an idea of the expected time to run each operation.
- *Standard Deviation*: Not only it is important to know the average running time, but also its volatility. A small standard deviation will tell us how predictable and reliable an operating system is in executing an operation within a certain amount of time. A large standard deviation will tell us that an operating system is not very predictable or reliable with expectations.
- *Maximum and Minimum Running Times*: This measurement is directly related with the standard deviation. The narrower this range, the better.
- *Margin of Error*: All statistical measurements carry some level of inaccuracy, thus it is important to measure how certain we are regarding the precision of the results. We are adopting 99% for the confidence level of all performed experiments.
- *Histograms*: Sometimes an image (chart) goes a long way in understanding a scenario. We will be able to visually compare each operating system side-by-side.

- *Algebra of Sets*: Portions of the sample data of each operating system will, in some cases, intersect with the others. We will need to know what elements are contained only in sample *A*, what elements are contained only in sample *B*, and what elements are an intersection of sample *A* and sample *B*.
- *Chi-Square Goodness-of-Fit Test*: When the differences in performance among the operating systems are not evident to the naked eye, we will use χ^2 to elucidate the differences and obtain statistical significance to accept or reject the hypothesis that the behaviors of the operating systems are the same.

Analysis of the Data

All time units, throughout all the experiments, are reported in milliseconds. However, for the programs written in C, elapsed times for Linux and OpenSolaris were measured in microseconds (10^{-3} milliseconds). A conversion to milliseconds was necessary, but it had no impact in the accuracy since the collected data has a much higher resolution than the converted data.

As a convention of nomenclature, P_j and W_j will represent Posix and Win32, respectively. Posix will encompass both Linux and OpenSolaris (C and Java) and Win32 will cover Windows (C and Java). The subscript j will represent an index to an item in a referred table.

Linear - C

We first start taking a look at the sample data from running the reference program written in C and executing the operations in a linear way.

Number of Iterations	Concurrent Threads	Total Running Time (milliseconds)		
		Linux	OpenSolaris	Windows
100	1	4,388	4,002	48,719

Table 1. *Linear - C*. Number of threads and total running time.

Table 1 shows that within this experiment, OpenSolaris was faster than Linux by a small margin, and that both Posix based operating systems were significantly faster than Windows (approximately 1 order of magnitude).

In order to understand better what happened and how each operation contributed in the composition of those times, we have to explore the sample data and learn what it says.

Time Measurements					
		Iteration Time	Counter Increment	Equation Calculation	File Replication
Average	Linux	43.86	7.55	23.35	12.51
	OpenSolaris	39.74	7.23	23.64	8.84
	Windows	461.43	42.68	22.52	372.04
Standard Deviation	Linux	3.87	0.23	0.00	0.02
	OpenSolaris	0.89	0.19	0.08	0.83
	Windows	467.69	6.92	7.85	459.72
Maximum	Linux	82.10	9.29	23.36	12.58
	OpenSolaris	46.94	8.92	24.41	16.06
	Windows	2,340.00	47.00	32.00	2,275.00
Minimum	Linux	43.03	7.09	23.34	12.46
	OpenSolaris	39.34	7.11	23.62	8.52
	Windows	125.00	31.00	15.00	47.00
Margin of Error	Linux	0.996	0.059	0.001	0.005
	OpenSolaris	0.229	0.048	0.021	0.214
	Windows	120.478	1.784	2.023	118.423

Table 2. *Linear* - C. Summary Statistics.

From Table 2 we can learn about some interesting statistics that can help us understand better how each operating system behaved during the execution of this experiment. On the average execution times (μ) Linux and OpenSolaris had equivalent performance for operations with integers and floating point and math. There was little, if any, difference in operations with integers ($|\mu_{\text{Linux}} - \mu_{\text{OpenSolaris}}| > \sigma_{\text{Linux}} + \sigma_{\text{OpenSolaris}}$). And practically no difference in operations with floating point and math, with Linux being a bit faster. For the I/O operations OpenSolaris was, on average, about 41% faster than Linux.

One thing worth noticing here is that both Linux and OpenSolaris were very predictable in their respective expected running times. The standard deviations were very narrow, which means to say that both operating systems were very consistent in their execution times.

When comparing Linux and/or OpenSolaris to Windows it is clear from the sample data that the formers were much faster than the latter in operations with integers and I/O. Regarding operations with floating point and math, although the average running time appears to have been faster, there is no statistical evidence to confirm that since the standard deviation is much larger than the time difference between the averages ($\sigma_{\text{Windows}} > |\mu_{\text{Linux or OpenSolaris}} - \mu_{\text{Windows}}|$). Windows overall showed very large standard deviation numbers in all categories being meas-

ured. This volatility suggests that the expected running time for operations being executed on Windows is by far less accurate than Linux and OpenSolaris.

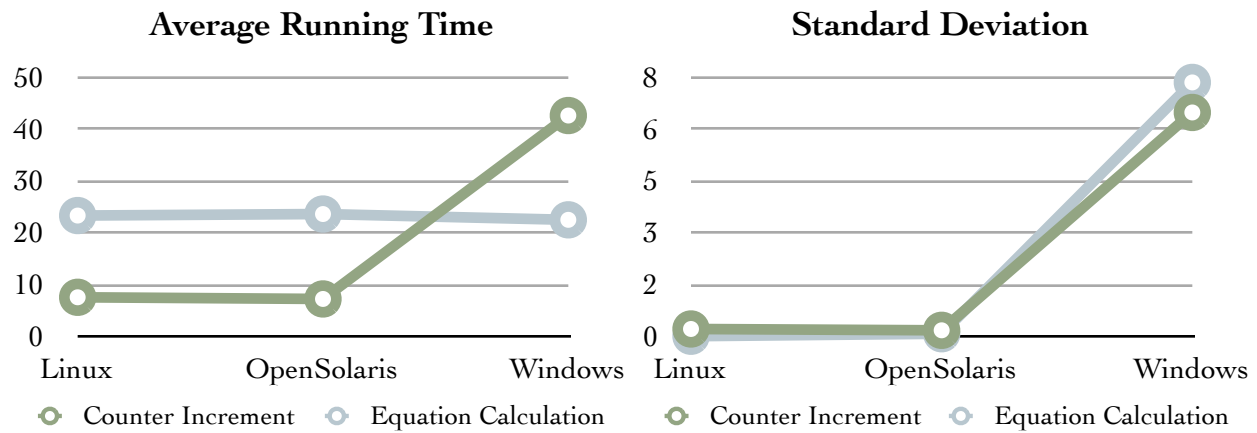


Chart 1. *Linear - C.* Average Running Time for operations with integers and floating point and math.

Chart 2. *Linear - C.* Standard Deviation for operations with integers and floating point and math.

On Chart 1 and also on Table 2 we see that the average time for the floating point and math operations on Windows were 22.52 milliseconds versus 23.35 milliseconds on Linux and 23.64 milliseconds on OpenSolaris. However, due to the higher Windows volatility further analysis of the data is necessary to understand this better.

Since Linux and OpenSolaris had very similar performances on executing those operations, for simplicity we can safely assume that they had the same performance and run the comparison only against one of the two operating systems and Windows. Thus we will need to analyze the sample data and see how many times Linux was faster than Windows, how many times they had equivalent performance, and how many times Linux was slower than Windows.

Table 3 gives us the frequency analysis of the sample data for Linux, OpenSolaris and Windows in regards to floating point and math operations:

Equation Calculation				
Bin Index	Time Ranges	Linux Frequency	OpenSolaris Frequency	Windows Frequency
1	15.00	0	0	22
2	16.80	0	0	34
3	18.60	0	0	0
4	20.40	0	0	0
5	22.20	0	0	0
6	24.00	100	99	0
7	25.80	0	1	0
8	27.60	0	0	0
9	29.40	0	0	0
10	33.00	0	0	44

Table 3. *Linear - C.* Frequency analysis of the running times for operations with floating point and math for Linux, OpenSolaris, and Windows.

Plotting the data in a histogram chart helps us visualize the distribution of execution times per time slot.

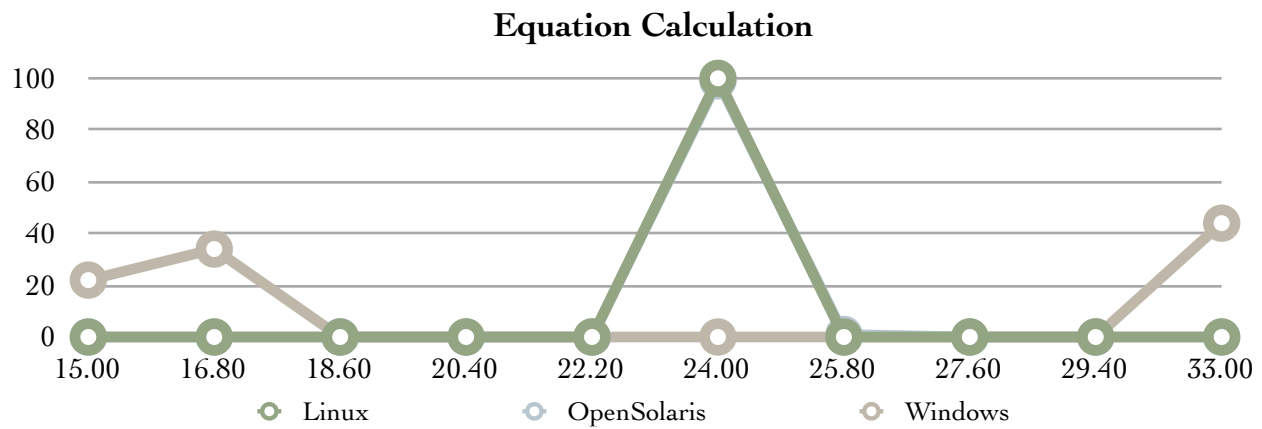


Chart 3. *Linear - C.* Histogram for operations with floating point and math.

Let's define W as the set containing all samples from running the experiment on Windows, and P as the set containing all samples from running the experiment on a Posix environment (Linux or OpenSolaris). For this case, P is representing the sample from running the experiment on Linux.

On Table 3 we classified the sample running times for floating point and math into time ranges. Assuming that any given two time samples that fall within the limits of a time range interval $[TR_{t-1}, TR_t]$ are said to have equivalent performance.

We start by calculating the subset of the times the performance of the two operating systems were equivalent.

$$E = P \cap W$$

Then we calculate subset of the times Linux was slower than Windows.

$$S = W - P \cap W$$

And finally we calculate the subset of the times Linux was faster than Windows.

$$F = P - P \cap W$$

For simplicity of reading, it is probably better to show the numbers in percentage format, where the sum of $E + S + F = 100\%$. In order to do that, we also need to calculate the set of samples that contained both in the Linux and Windows sets.

$$P \cup W = P + W - P \cap W$$

Now we can calculate E, S, F in terms of percentage.

$$E\% = \frac{1}{\text{count}(P \cap W)} \sum_{j=1}^{10} E_j$$

$$S\% = \frac{1}{\text{count}(P \cap W)} \sum_{j=1}^{10} S_j$$

$$F\% = \frac{1}{\text{count}(P \cap W)} \sum_{j=1}^{10} F_j$$

Since we classified the running times of the samples into time ranges, the algebra of sets for each time range can be calculated using the following cases.

$$E_j = \min(P_j, W_j)$$

$$S_j = \begin{cases} W_j - P_j & \text{if } (P_j < W_j) \wedge (\sum_{i=1}^j P_i < 100) \\ 0 & \text{if } (P_j > W_j) \wedge (\sum_{i=1}^j P_i > 0) \wedge (\sum_{i=1}^j W_i < 100) \\ P_j - (P_j \cap W_j) & \text{otherwise} \end{cases}$$

$$F_j = \begin{cases} P_j - W_j & \text{if } P_j > W_j \\ 0 & \text{if } ((P_j > W_j) \wedge (\sum_{i=1}^j P_i = 100)) \vee (P_j = W_j) \vee (\sum_{i=1}^j P_i < 100) \\ W_j - P_j & \text{otherwise} \end{cases}$$

Equation Calculation			
Time Ranges	Linux Faster than Windows	Linux Equal to Windows	Linux Slower than Windows
15.00	0	0	22
16.80	0	0	34
18.60	0	0	0
20.40	0	0	0
22.20	0	0	0
24.00	100	0	0
25.80	0	0	0
27.60	0	0	0
29.40	0	0	0
33.00	44	0	0
Totals	144	0	56
	72.00%	0.00%	28.00%

Table 4. *Linear - C.* Performance comparison between Linux and Windows on operations with floating point and math.

From the results shown in Table 4 we can see that Linux was in fact faster than Windows 72% of the time. They did not intersect at any point, and Windows was faster than Linux 28% of the time.

Another point worth looking into is the performance related to I/O operations. OpenSolaris and its Zettabyte File System (ZFS) took the lead in this case. Windows and its NT File System (NTFS) appear to be one of the weakest points of the operating system.

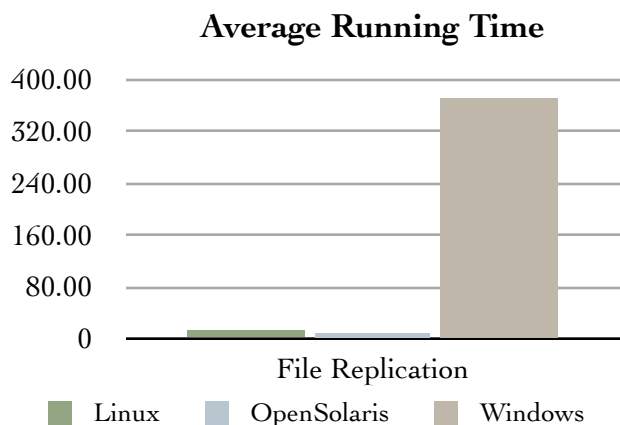


Chart 4. *Linear - C.* Average running time for I/O operations.

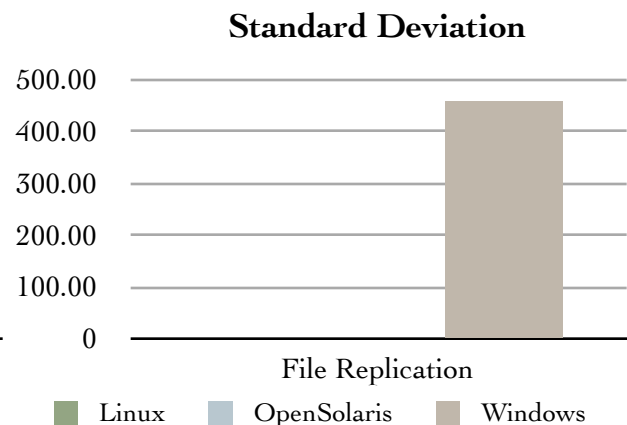


Chart 5. *Linear - C.* Standard deviation for I/O operations.

Not only the Windows average running time was considerably slower than OpenSolaris and Linux, as we can see in Chart 4, but also the space occupied on disk by the files themselves was significantly larger than OpenSolaris using ZFS and Linux using ext4. Table 5 shows the disk spaced used for the I/O operations by each of the file systems.

	Linux (ext4)	OpenSolaris (ZFS)	Windows (NTFS)
Space Used on Disk (MB)	335	335	351

Table 5. *Linear - C.* Disk spaced used per filesystem.

Each execution of any of the programs in this experiment creates a directory and copies the GPL 100 times into that directory for each iteration, and since each program execution iterates 100 times, at the end of each experiment there will be 10,100 items (10,000 files and 100 directories). Doing the math to calculate NTFS' overhead we have:

$$\text{Total Overhead} = 351 \text{ MB} - 335 \text{ MB} = 16 \text{ MB} (16,777,216 \text{ bytes})$$

$$\text{Average Overhead per Item} = \frac{16,777,216 \text{ bytes}}{10,100 \text{ items}} \approx 1,661 \text{ bytes per item}$$

The Windows filesystem seems to need to save a lot more overhead information to store the same amount of data⁵.

Linear - Java

The second set of data comes from running the reference program written in Java, also executing the operations in a linear way.

Number of Iterations	Concurrent Threads	Total Running Time (milliseconds)		
		Linux	OpenSolaris	Windows
100	1	257,836	245,548	280,363

Table 6. *Linear - Java.* Number of threads and total running time.

The differences among the operating systems are not as accentuated as in the *Linear - C* case, nonetheless we can clearly perceive that OpenSolaris and Linux performed better than Windows, with OpenSolaris having the best overall performance among the operating systems for this case. Let's take a look into the details of the experiment.

⁵ This behavior was observed across all experiments.

Time Measurements					
		Iteration Time	Counter Increment	Equation Calculation	File Replication
Average	Linux	2,570.45	2,357.55	156.18	56.54
	OpenSolaris	2,455.04	2,311.03	87.86	56.03
	Windows	2,794.69	2,495.54	169.15	128.74
Standard Deviation	Linux	50.60	31.88	16.51	10.31
	OpenSolaris	33.00	22.85	1.61	10.22
	Windows	83.51	81.48	5.78	15.65
Maximum	Linux	2,693.00	2,425.00	184.00	130.00
	OpenSolaris	2,751.00	2,497.00	103.00	147.00
	Windows	2,948.00	2,637.00	172.00	203.00
Minimum	Linux	2,484.00	2,293.00	141.00	48.00
	OpenSolaris	2,430.00	2,288.00	87.00	52.00
	Windows	2,667.00	2,371.00	156.00	109.00
Margin of Error	Linux	13.03	8.21	4.25	2.66
	OpenSolaris	8.50	5.89	0.42	2.63
	Windows	21.51	20.99	1.49	4.03

Table 7. Linear - Java. Summary Statistics.

Table 7 shows that OpenSolaris performed better across the board in the Java linear experiment. It had the lowest average running time and the least volatility in all categories being measured. The histogram of the iteration times is shown in Chart 6.

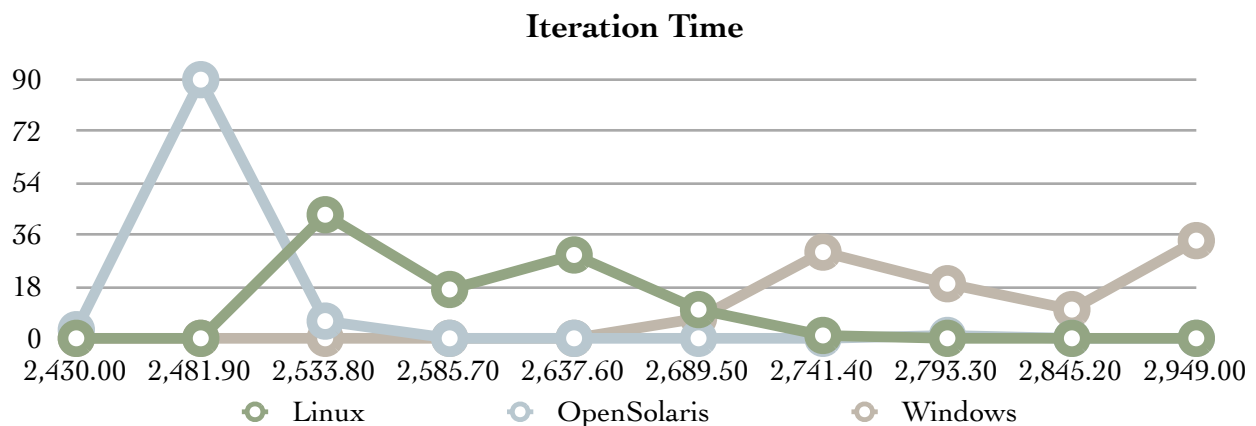


Chart 6. Linear - Java. Histogram for iteration running times.

One thing we can notice from Table 7 is that although Linux was on average faster than Windows in floating point and math operations, it had a higher volatility in its execution time. Let's take a closer look at how this volatility affects the results and compare it to the same operations on Windows.

Equation Calculation				
Bin Index	Time Ranges	Linux Frequency	OpenSolaris Frequency	Windows Frequency
1	87.00	0	31	0
2	96.80	0	68	0
3	106.60	0	1	0
4	116.40	0	0	0
5	126.20	0	0	0
6	136.00	0	0	0
7	145.80	54	0	0
8	155.60	2	0	0
9	165.40	8	0	16
10	185.00	36	0	84

Table 8. *Linear - Java*. Frequency analysis of the running times for operations with floating point and math for Linux, OpenSolaris, and Windows.

The data shown on Table 8 suggests that there are a significant number of performance overlaps between Linux and Windows. First we need to know if there is statistical evidence showing that the two sets of sample data are different. We will use Chi-Square Goodness-of-Fit Test (χ^2) to determine that.

Our null hypothesis is that the two sample data are equal (or very similar) and the alternative hypothesis is that they are not similar at $\alpha = 0.05$ (5%).

$$H_0 : P = W$$

$$H_1 : P \neq W$$

Let p_{ij} be the probabilities of the events, where i is the index for the time range bin and j is the index for the operating system. Each program run is iterated $n = 100$ times. Also, let F_{ij} be the frequencies of each event. The sum

$$Q = \sum_{i=1}^{10} \sum_{j=1}^2 \frac{(F_{ij} - np_{ij})^2}{np_{ij}}$$

is approximately the chi-square with a maximum of

Time range bins $- 1 +$ Operating systems $- 1 = (10 - 1) + (2 - 1) = 10$ degrees of freedom.

Chi-Square Test - Equation Calculation		
p	np	q
0.000	0.0	
0.000	0.0	
0.000	0.0	
0.000	0.0	
0.000	0.0	
0.000	0.0	
0.270	27.0	54.000
0.010	1.0	2.000
0.120	12.0	2.667
0.600	60.0	19.200
	Q=	77.867
Expected Chi-Square (0.05, 4) =		9.488

Table 9. *Linear - Java.* Linux and Windows Chi-Square Goodness-of-Fit Test calculation for operations with floating point and math.

Table 9 has the calculated Chi-Square (Q) and the expected ($\chi_{0.05}^2(4)$) Chi-Square with 4 degrees of freedom.

$$p = p_{i1} + p_{i2} = \begin{bmatrix} \frac{54}{200} \\ \frac{2}{200} \\ \frac{8}{200} \\ \frac{36}{200} \end{bmatrix} + \begin{bmatrix} \frac{0}{200} \\ \frac{0}{200} \\ \frac{16}{200} \\ \frac{84}{200} \end{bmatrix} = \begin{bmatrix} 0.27 \\ 0.01 \\ 0.12 \\ 0.60 \end{bmatrix}$$

$$n \cdot p = 100 \cdot \begin{bmatrix} 0.27 \\ 0.01 \\ 0.12 \\ 0.60 \end{bmatrix} = \begin{bmatrix} 27 \\ 1 \\ 12 \\ 60 \end{bmatrix}$$

$$q = q_{i1} + q_{i2} = \begin{bmatrix} \frac{(54-27)^2}{27} \\ \frac{(2-1)^2}{1} \\ \frac{(8-12)^2}{12} \\ \frac{(36-60)^2}{60} \end{bmatrix} + \begin{bmatrix} \frac{(0-27)^2}{27} \\ \frac{(0-1)^2}{1} \\ \frac{(16-12)^2}{12} \\ \frac{(84-60)^2}{60} \end{bmatrix}$$

$$Q = 77.867$$

Since $Q = 77.867 > 9.488 = \chi_{0.05}^2(4)$, the null hypothesis H_0 is rejected. Which means that there is no statistical evidence that shows that the two samples of data are equal.

Since now we know that the two data samples are not equal, we can compare their time ranges in executing the operations.

Time Range Comparison - Equation Calculation			
Range	Linux	OpenSolaris	Windows
≤ 96.80	0%	99%	0%
> 96.80 and ≤ 145.80	54%	1%	0%
> 145.80	46%	0%	100%

Table 10. *Linear - Java.* Time range comparison for operations with floating point and math.

Table 10 shows us that 54% of the time Linux executed the floating point and math operations in less than or equal to 145.80 milliseconds. In contrast Windows executed 100% of the operations in more than 145.80 milliseconds. The evidence shows that even though Linux had a higher volatility than Windows in the expected execution time, overall it performed better than Windows the majority of the time.

OpenSolaris was the fastest executing the I/O operations and also had the least volatility, as we can see on Charts 7 and 8, respectively. Windows was the slowest among the operating systems and had the highest volatility.

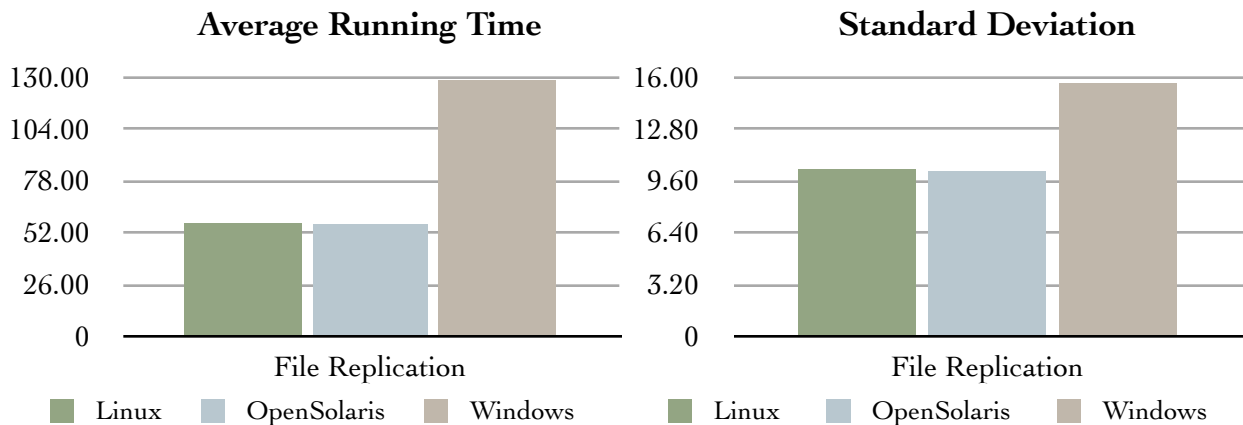


Chart 7. *Linear - Java.* Average running time for I/O operations. **Chart 8.** *Linear - Java.* Standard deviation for I/O operations.

Threads - C

This experiment is the first one to take advantage of the availability of multi-cores and the multithreading capabilities of the operating systems. The sample data comes from running each operation (integers, floating point and math, and I/O) on its own thread, achieving parallel processing on the same machine.

Number of Iterations	Concurrent Threads	Total Running Time (milliseconds)		
		Linux	OpenSolaris	Windows
100	3	3,339	2,388	42,167

Table 11. *Threads - C.* Number of threads and total running time.

Comparing Table 11 and Table 1 we notice that all operating systems were able to benefit from the multithreaded way of executing operations, and on average, each iteration was faster when compared to the respective linear programs. Moreover, the total time needed to finish each experiment was smaller.

Time Measurements					
		Iteration Time	Counter Increment	Equation Calculation	File Replication
Average	Linux	33.36	8.54	27.28	15.09
	OpenSolaris	23.85	7.32	23.71	10.23
	Windows	421.21	57.74	25.87	401.46
Standard Deviation	Linux	6.05	1.34	2.65	4.51
	OpenSolaris	0.18	0.36	0.16	1.85
	Windows	436.02	7.86	7.43	435.84
Maximum	Linux	54.93	12.92	36.22	54.78
	OpenSolaris	24.77	8.32	24.53	22.15
	Windows	2,278.00	78.00	32.00	2,262.00
Minimum	Linux	23.63	6.84	23.37	12.50
	OpenSolaris	23.74	6.66	23.64	8.77
	Windows	62.00	46.00	15.00	46.00
Margin of Error	Linux	1.56	0.35	0.68	1.16
	OpenSolaris	0.05	0.09	0.04	0.48
	Windows	112.32	2.02	1.91	112.27

Table 12. *Threads - C.* Summary Statistics.

One thing worth noticing is that on average, each individual operation took longer to run than their equivalent linear programs. However, since the operations are being executed in parallel, the total time of an iteration is approximately the time the slowest operation takes to complete.

Windows had the average running time for operations with floating points and math smaller than Linux, but it also had a higher volatility in executing those operations. Once more we have to look deeper into the sample data to understand this better.

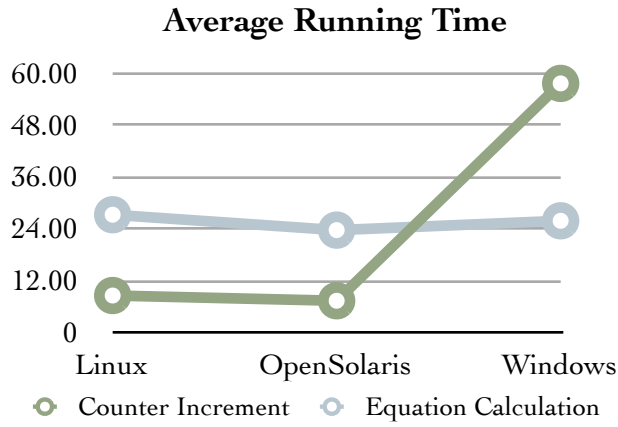


Chart 9. *Threads - C.* Average Running Time for operations with integers and floating point and math.

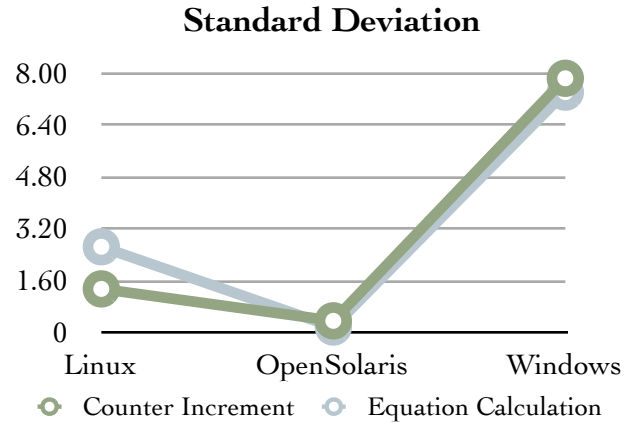


Chart 10. *Threads - C.* Standard Deviation for operations with integers and floating point and math.

Looking at Chart 11, we cannot tell whether the two sample data histograms (Linux and Windows) are equal or different.

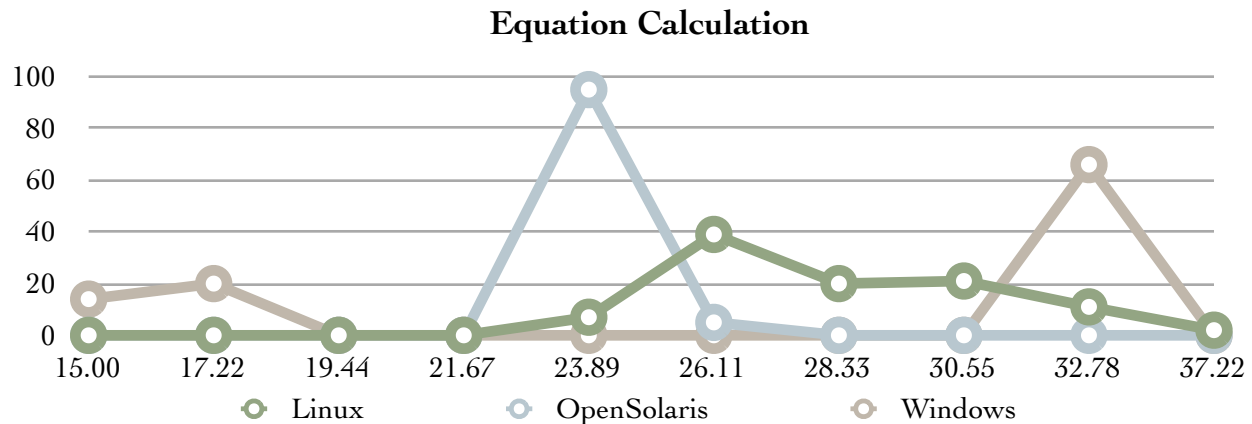


Chart 11. *Threads - C.* Histogram for operations with floating point and math.

Nevertheless, we should perform a formal statistical analysis to determine whether the two sample data are equal. If we go ahead and propose a hypothesis that the two are equal and an alternative hypothesis that they are not equal:

$$H_0 : P = W$$

$$H_1 : P \neq W$$

We can calculate the Chi-Square Goodness-of-Fit Test as follows

Chi-Square Test - Equation Calculation		
p	np	q
0.070	7.0	14.000
0.100	10.0	20.000
0.000	0.0	
0.000	0.0	
0.035	3.5	7.000
0.195	19.5	39.000
0.100	10.0	20.000
0.105	10.5	21.000
0.385	38.5	39.286
0.010	1.0	2.000
	Q=	162.286
Expected Chi-Square (0.05, 8) =		15.507

Table 13. *Threads* - C. Linux and Windows Chi-Square Goodness-of-Fit Test calculation for operations with floating point and math.

And reject the null hypothesis that the two samples of data are equal, since $Q = 162.286 > 15.507 = \chi_{0.05}^2(8)$.

From the frequency analysis table shown on Table 14, we can calculate how many times each operating system was faster than the other and how many times they had overlapping equivalent performance.

Equation Calculation			
Time Ranges	Linux Faster than Windows	Linux Equal to Windows	Linux Slower than Windows
15.00	0	0	14
17.22	0	0	20
19.44	0	0	0
21.67	0	0	0
23.89	7	0	0
26.11	39	0	0
28.33	20	0	0
30.55	21	0	0
32.78	0	11	55
37.22	0	0	2
Totals	87	11	91
	46.03%	5.82%	48.15%

Table 14. *Threads - C.* Performance comparison between Linux and Windows on operations with floating point and math.

Linux and Windows were faster than the other about the same amount of times, however with the high volatility on Windows execution times, we can only conclude that there is no statistical evidence to say in this case, whether one was faster than the other in executing operations with floating point and math.

OpenSolaris was again the fastest executing the I/O operations and had the least volatility. Windows continued to be the slowest among the operating systems and also the one with the highest volatility.

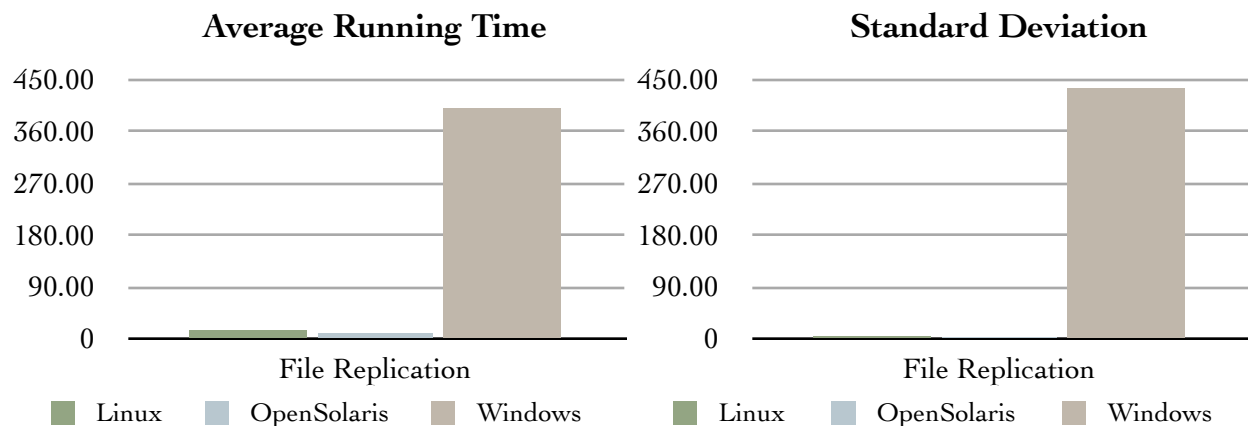


Chart 12. *Threads - C.* Average running time for I/O operations. **Chart 13.** *Threads - C.* Standard deviation for I/O operations.

Threads - Java

This is the second experiment to take advantage of executing the operations in parallel using multithreading capabilities of the machine and operating systems. The sample data is originated from running the multithreaded program in Java and utilizing the virtual machine (JVM).

Number of Iterations	Concurrent Threads	Total Running Time (milliseconds)		
		Linux	OpenSolaris	Windows
100	3	244,511	231,800	244,905

Table 15. *Threads - Java.* Number of threads and total running time.

Again, as in the previous analysis (*Threads - C*), we can see that all operating systems benefited from running the operations in parallel on separated threads when compared to the linear experiment.

Time Measurements					
		Iteration Time	Counter Increment	Equation Calculation	File Replication
Average	Linux	2,444.71	2,443.51	184.13	88.96
	OpenSolaris	2,317.48	2,316.95	90.06	66.85
	Windows	2,448.28	2,444.54	172.24	146.33
Standard Deviation	Linux	45.96	46.17	32.12	29.27
	OpenSolaris	28.43	27.99	4.26	18.06
	Windows	76.84	68.88	6.99	29.01
Maximum	Linux	2,635.00	2,630.00	299.00	205.00
	OpenSolaris	2,518.00	2,512.00	124.00	189.00
	Windows	2,808.00	2,605.00	203.00	268.00
Minimum	Linux	2,358.00	2,358.00	145.00	53.00
	OpenSolaris	2,289.00	2,289.00	87.00	52.00
	Windows	2,371.00	2,371.00	156.00	109.00
Margin of Error	Linux	11.84	11.89	8.27	7.54
	OpenSolaris	7.32	7.21	1.10	4.65
	Windows	19.79	17.74	1.80	7.47

Table 16. *Threads - Java.* Summary Statistics.

As expected, we once more can see that although the average running time for each iteration was smaller than the equivalent java experiment running the operations in linear mode, each individual operation took longer to complete. However, even though the operations with integers in Linux and Windows may look like two exceptions, we cannot assert that since there is no

statistical evidence for such an assumption. Their respective standard deviations are larger than the difference between the average running times ($\sigma_{\text{integers}} > |\mu_{\text{integers linear}} - \mu_{\text{integers threaded}}|$).

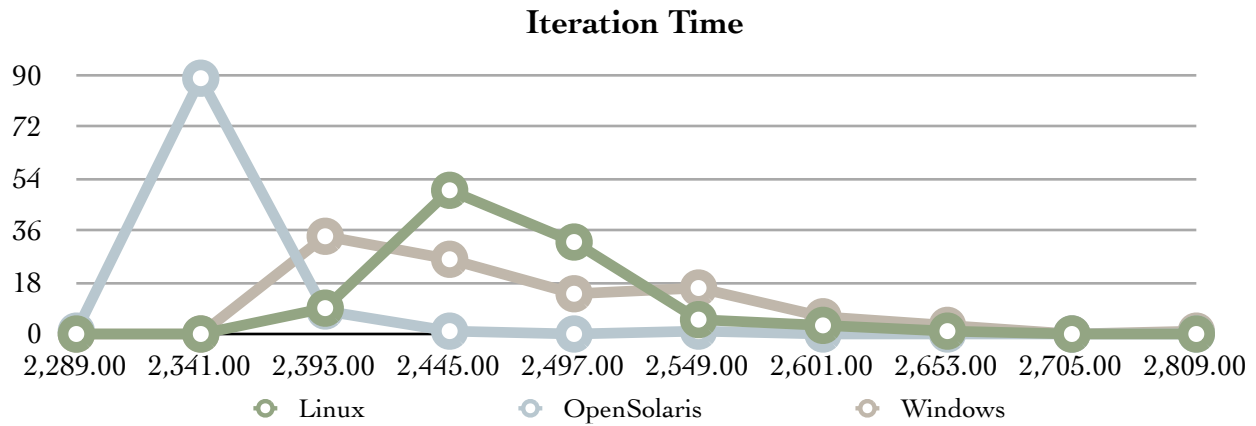


Chart 14. *Threads - Java.* Histogram for iteration running times.

Chart 14 shows us that there were lots of overlapping in performance, at least between Linux and Windows; OpenSolaris performed better than the other two operating systems in all metrics. We will need to take a closer look at the sample data and analyze it per measured category. Let's start looking at operations with integers.

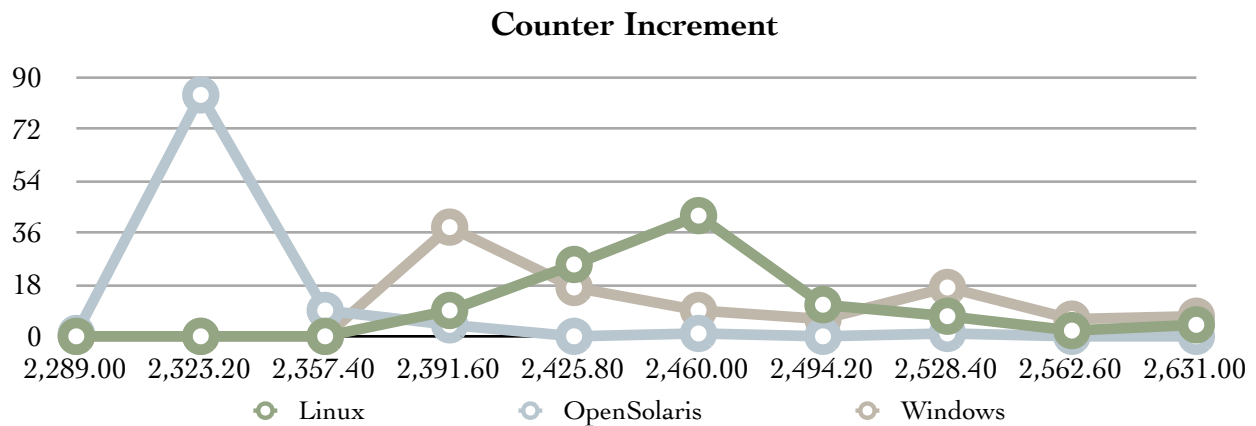


Chart 15. *Threads - Java.* Histogram for operations with integers.

Counter Increment				
Bin Index	Time Ranges	Linux Frequency	OpenSolaris Frequency	Windows Frequency
1	2,289.00	0	1	0
2	2,323.20	0	84	0
3	2,357.40	0	9	0
4	2,391.60	9	4	38
5	2,425.80	25	0	17
6	2,460.00	42	1	9
7	2,494.20	11	0	6
8	2,528.40	7	1	17
9	2,562.60	2	0	6
10	2,631.00	4	0	7

Table 17. *Threads - Java.* Frequency analysis of the running times for operations with integers for Linux, OpenSolaris, and Windows.

There is some similarity between the Linux and Windows histograms, however, to be sure we will propose a null hypothesis that the two curves are similar and calculate the Chi-Square to accept or reject the null hypothesis.

Chi-Square Test - Counter Increment		
p	np	q
0.000	0.0	
0.000	0.0	
0.000	0.0	
0.235	23.5	17.894
0.210	21.0	1.524
0.255	25.5	21.353
0.085	8.5	1.471
0.120	12.0	4.167
0.040	4.0	2.000
0.055	5.5	0.818
	Q=	49.226
Expected Chi-Square (0.05, 7) =		14.067

Table 18. *Threads - Java.* Linux and Windows Chi-Square Goodness-of-Fit Test calculation for operations with integers.

We have to reject the null hypothesis that the two curves are similar because the calculated Chi-Square is much larger than the expected Chi-Square $Q = 49.226 > 14.067 = \chi_{0.05}^2(7)$.

Now that we know that there is no statistical evidence to say that the curves are similar, the next step is to calculate how many times each operating system was faster than the other and how many times they had equivalent performance.

Counter Increment			
Time Ranges	Linux Faster than Windows	Linux Equal to Windows	Linux Slower than Windows
2,289.00	0	0	0
2,323.20	0	0	0
2,357.40	0	0	0
2,391.60	0	9	29
2,425.80	8	17	0
2,460.00	33	9	0
2,494.20	5	6	0
2,528.40	0	7	10
2,562.60	0	2	4
2,631.00	3	4	0
Totals	49	54	43
	33.56%	36.99%	29.45%

Table 19. *Threads - Java.* Performance comparison between Linux and Windows on operations with integers.

The comparison on Table 19 shows that Linux and Windows had a large number of performance intersections (36.99% of the time), Windows was faster than Linux in 29.45% of the time, and Linux was faster than Windows 33.56% of the time.

Another performance analysis we can make is the time range comparison. We can see on Table 20 that 76% of the time Linux completed the operations with integers between 2,323 milliseconds and 2,460 milliseconds versus 64% of the time for Windows.

Time Range Comparison - Counter Increment			
Range	Linux	OpenSolaris	Windows
<= 2,323.20	0%	85%	0%
> 2,323.20 and <= 2,460.00	76%	14%	64%
> 2,460.00	24%	1%	36%

Table 20. *Threads - Java.* Time range comparison for operations with integers.

Our next analysis is regarding the operations with floating point and math. Chart 16 and Table 21 show us the histogram distribution for these operations.

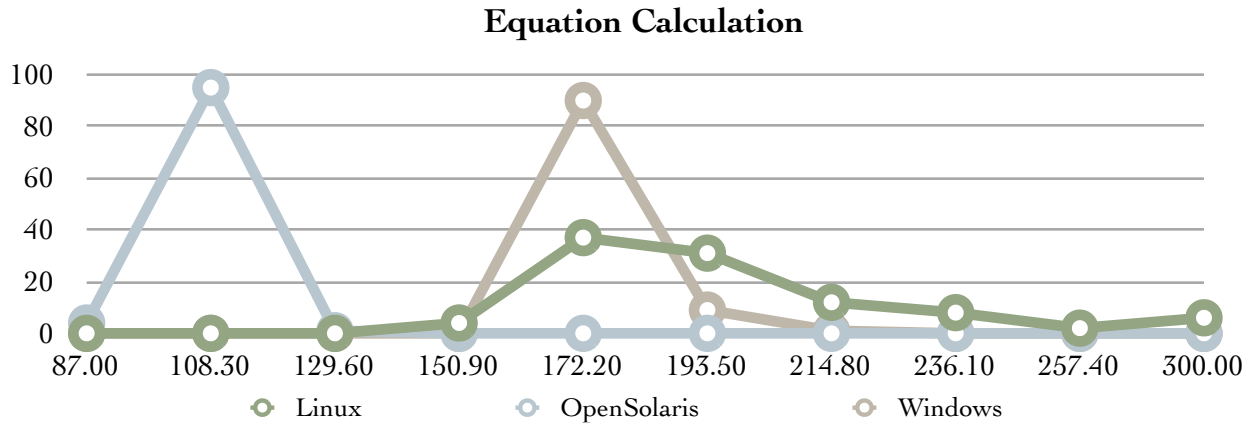


Chart 16. *Threads - Java.* Histogram for operations with floating point and math.

Equation Calculation				
Bin Index	Time Ranges	Linux Frequency	OpenSolaris Frequency	Windows Frequency
1	87.00	0	4	0
2	108.30	0	95	0
3	129.60	0	1	0
4	150.90	4	0	0
5	172.20	37	0	90
6	193.50	31	0	9
7	214.80	12	0	1
8	236.10	8	0	0
9	257.40	2	0	0
10	300.00	6	0	0

Table 21. *Threads - Java.* Frequency analysis of the running times for operations with floating point and math for Linux, OpenSolaris, and Windows.

Windows, did not perform as well as OpenSolaris. However there was an exception for one case, among 18 different measurements in total, where it performed better than Linux. If we analyze the time ranges we can see that in 90% of the time the former executed the floating point and math operations in less than 172 milliseconds versus 41% of the time for the latter.

Time Range Comparison - Equation Calculation			
Range	Linux	OpenSolaris	Windows
<= 172.20	41%	100%	90%
> 172.20	59%	0%	10%

Table 22. *Threads - Java.* Time range comparison for operations with floating point and math.

The next analysis is regarding the I/O operations. We can see on Chart 17 that OpenSolaris performed better among the three operating systems.

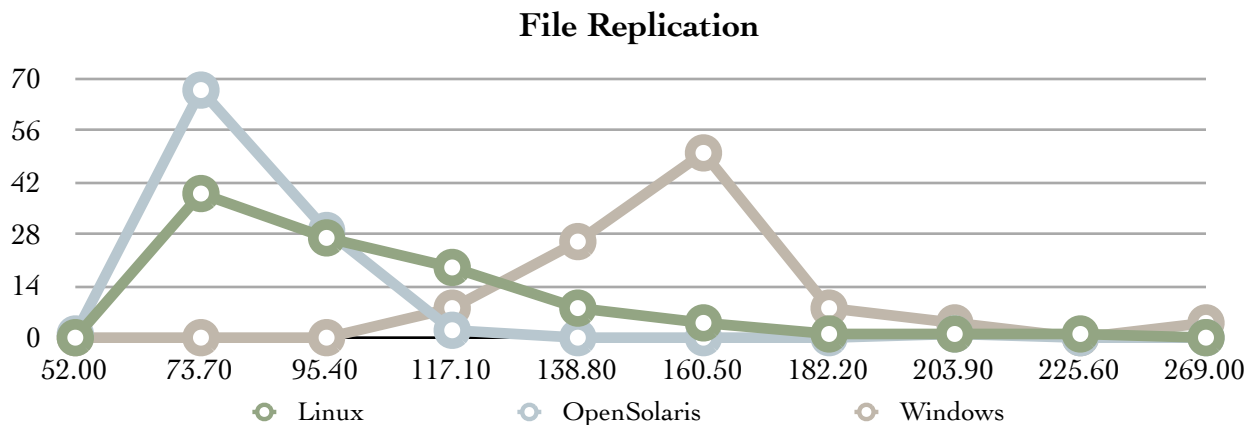


Chart 17. *Threads - Java.* Histogram for I/O operations.

As we can see on Charts 18 and 19, OpenSolaris had the smallest average running time and the lowest volatility. We also can notice that, for the first time, ext4 had a higher volatility than NTFS. However since the average running time was considerably faster, it compensated for the higher volatility.

$$\mu_{\text{Linux}} + \sigma_{\text{Linux}} < \mu_{\text{Windows}}$$

$$118.23 < 146.33$$

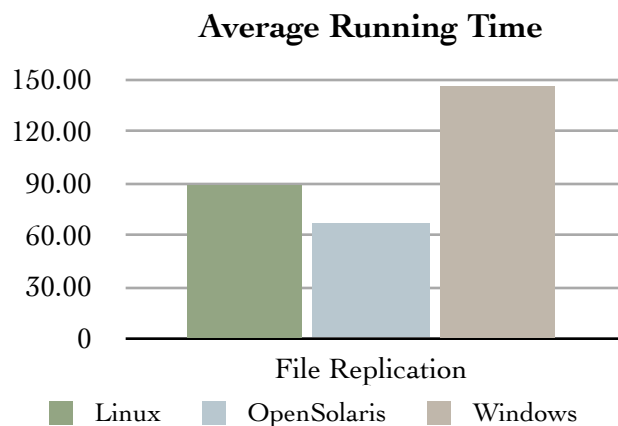


Chart 18. *Threads - Java.* Average running time for I/O operations.

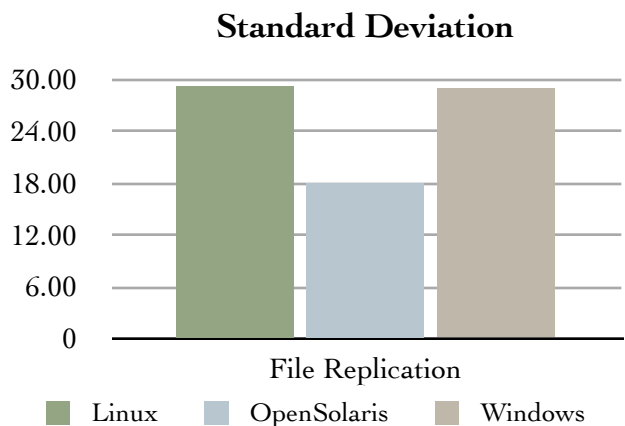


Chart 19. *Threads - Java.* Standard deviation for I/O operations.

Stress - C

Now we step into a territory where the machine is going to be overwhelmed with concurrent operations. There will be multiple threads competing for a limited amount of resources. Each operating system will have to manage which threads are to run and which threads are to wait idle for their time slot to run. This sample data was generated by running the stress programs written in C.

Number of Iterations	Concurrent Threads	Total Running Time (milliseconds)		
		Linux	OpenSolaris	Windows
100	103	169,660	388,446	230,991

Table 23. Stress - C. Number of threads and total running time.

The first thing we can see is that all three operating systems took a substantial larger amount of time to complete the operations. This was expected since the overhead generated by having to manage this enormous and chaotic concurrent environment requires a herculean effort from each operating system.

Time Measurements					
		Iteration Time	Counter Increment	Equation Calculation	File Replication
Average	Linux	1,696.57	14.93	1,695.78	26.34
	OpenSolaris	3,884.43	8.52	3,884.07	9.70
	Windows	2,309.45	60.50	2,288.82	99.61
Standard Deviation	Linux	337.15	5.45	337.14	9.93
	OpenSolaris	574.13	0.70	574.13	0.29
	Windows	1,451.30	6.36	1,448.19	90.66
Maximum	Linux	2,083.23	55.77	2,082.42	54.20
	OpenSolaris	5,739.59	11.32	5,739.23	10.17
	Windows	5,258.00	78.00	5,116.00	602.00
Minimum	Linux	1,035.38	9.08	1,034.74	14.40
	OpenSolaris	2,734.53	7.93	2,734.16	9.18
	Windows	1,029.00	46.00	1,014.00	62.00
Margin of Error	Linux	86.85	1.40	86.85	2.56
	OpenSolaris	147.90	0.18	147.90	0.08
	Windows	373.85	1.64	373.05	23.35

Table 24. Stress - C. Summary Statistics.

Another point that may catch our attention is that, for the first time, Windows finished in second place, ahead of OpenSolaris.

One unique element introduced in this experiment was the use of semaphores. We used the classical *Producer and Consumer* model to execute the operations with floating point and math. The introduction of semaphores wouldn't have been this burdensome if we did not have another set of 100 threads all competing to execute the operations with integers. A producer may have finished and signaled to the consumer, however, since the operating system has to time-slice manage the available resources and decide who is going to run next, the consumer may take a long time to start, execute the operations, and signal back to the producer that it can calculate another set of points.

Chart 20 shows the histogram with the running times of this experiment's iterations. Charts 21 and 22 show us the average running times and standard deviations for operations with integers and floating point and math.

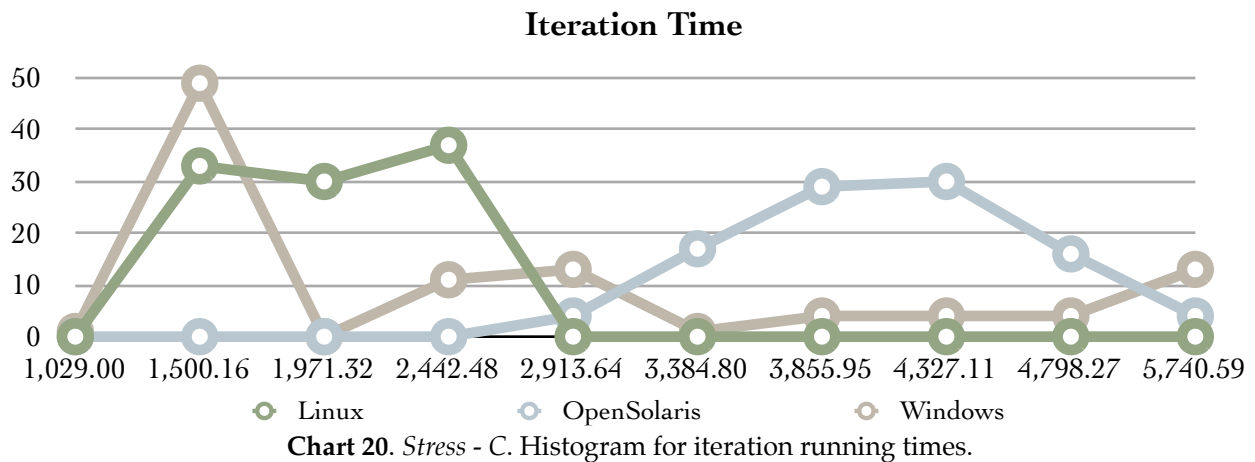


Chart 20. *Stress - C.* Histogram for iteration running times.

Under these circumstances of extreme stress this is the first time Linux performed better than OpenSolaris. Furthermore, Windows did not finish last, although it still had the highest volatility among the three operating systems.

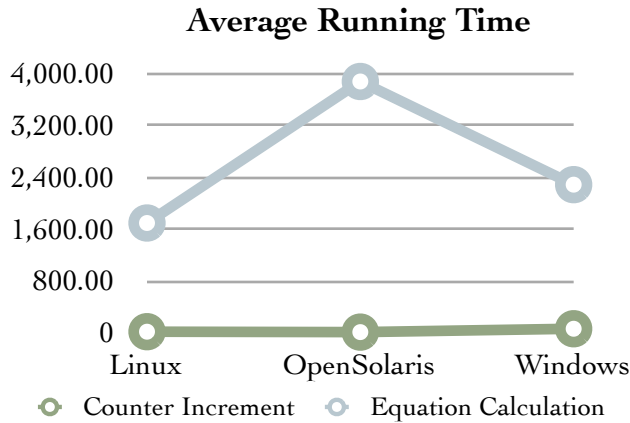


Chart 21. *Stress - C.* Average Running Time for operations with integers and floating point and math.

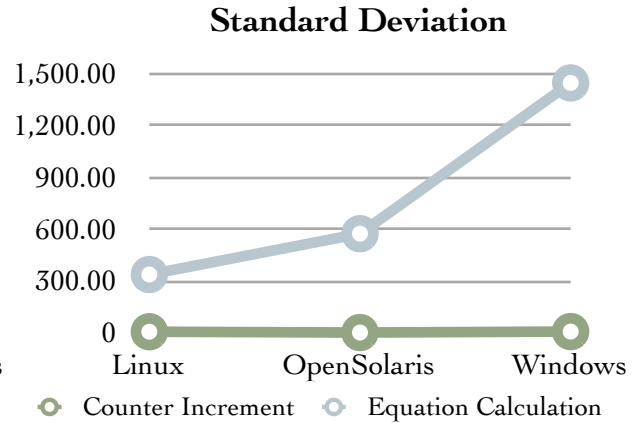


Chart 22. *Stress - C.* Standard Deviation for operations with integers and floating point and math.

After analyzing Charts 23, 24, and 25 we can see that the operations with floating point and math basically dictated the pace for this experiment and were responsible for consuming most of the time spent on each iteration. We can see that on the other two categories being measured OpenSolaris had better performance than Windows.

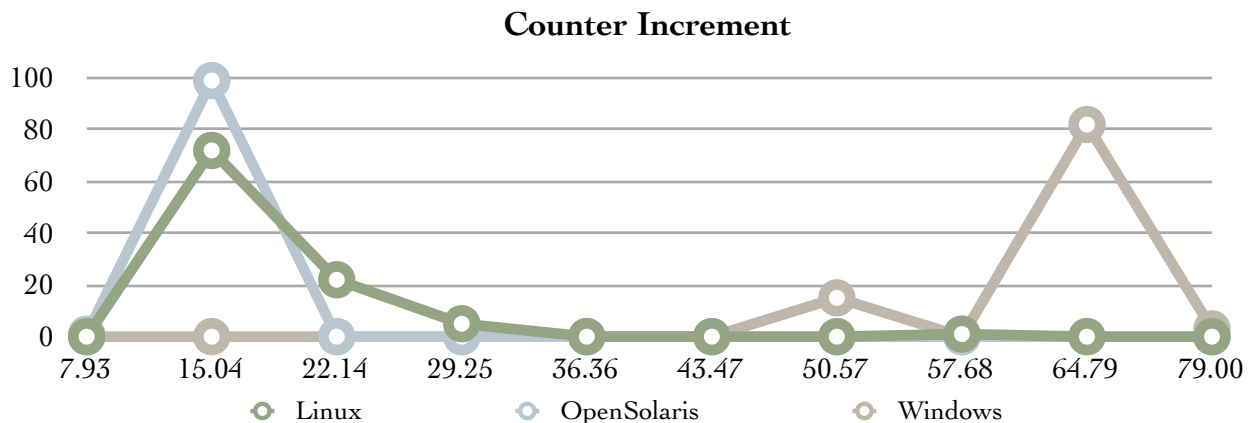


Chart 23. *Stress - C.* Histogram for operations with integers.

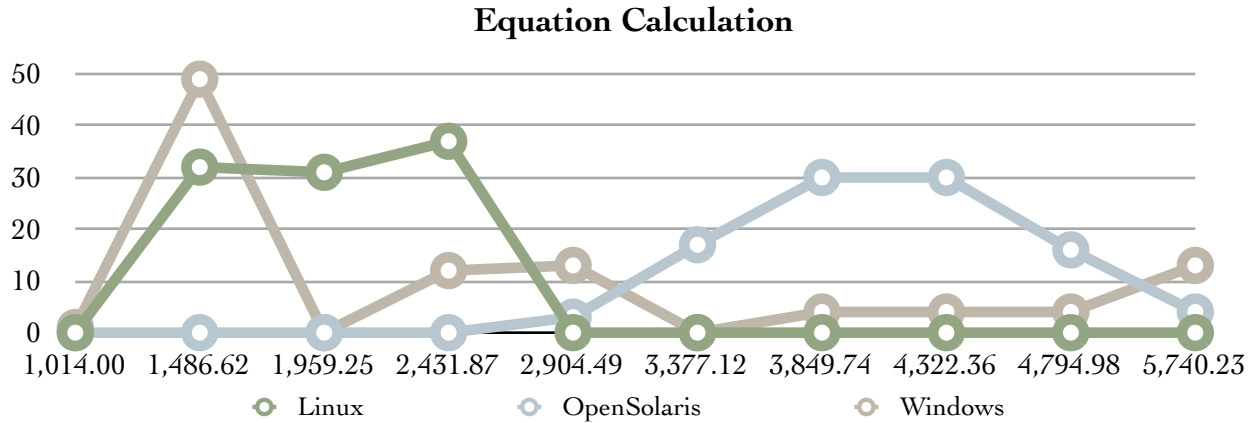


Chart 24. Stress - C. Histogram for operations with floating point and math.

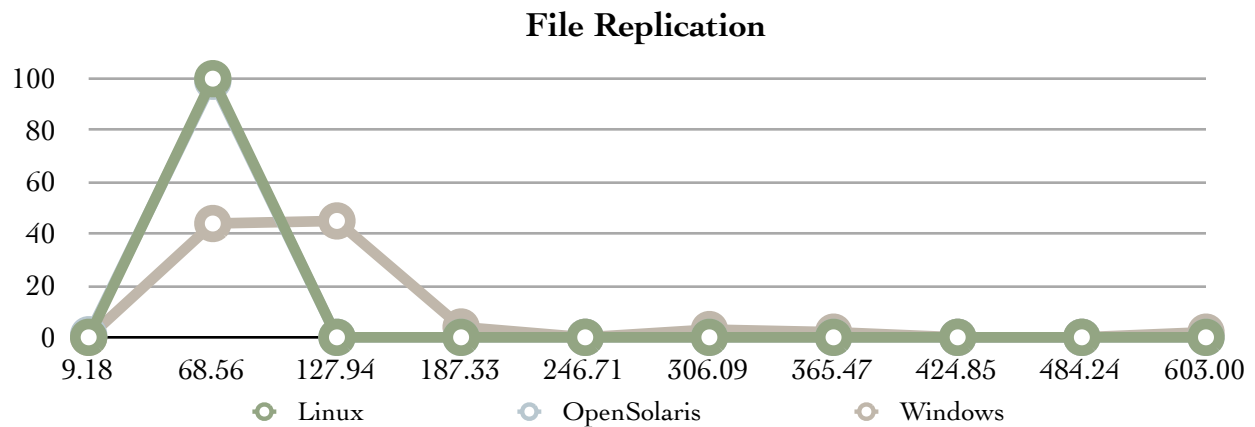


Chart 25. Stress - C. Histogram for I/O operations.

Table 25 shows us the time range comparison among the three operating systems regarding the performance of each to run the operations with floating point and math, while time-slice managing the execution of multiple concurrent threads.

Time Range Comparison - Equation Calculation			
Range	Linux	OpenSolaris	Windows
<= 1,959.25	63%	0%	50%
> 1,959.25 and <= 3,849.74	37%	50%	29%
> 3,849.74	0%	50%	21%

Table 25. Stress - C. Time range comparison for operations with floating point and math.

There must be some nuance in the semaphore thread scheduling algorithm in OpenSolaris, because it outperformed both Linux and Windows on the other two categories being measured. The C source code is identical for both Linux and OpenSolaris, compiled with the same options

and with the same version of gcc. The reason why OpenSolaris semaphores performed as such is a subject the author would like to explore further in another opportunity in the future.

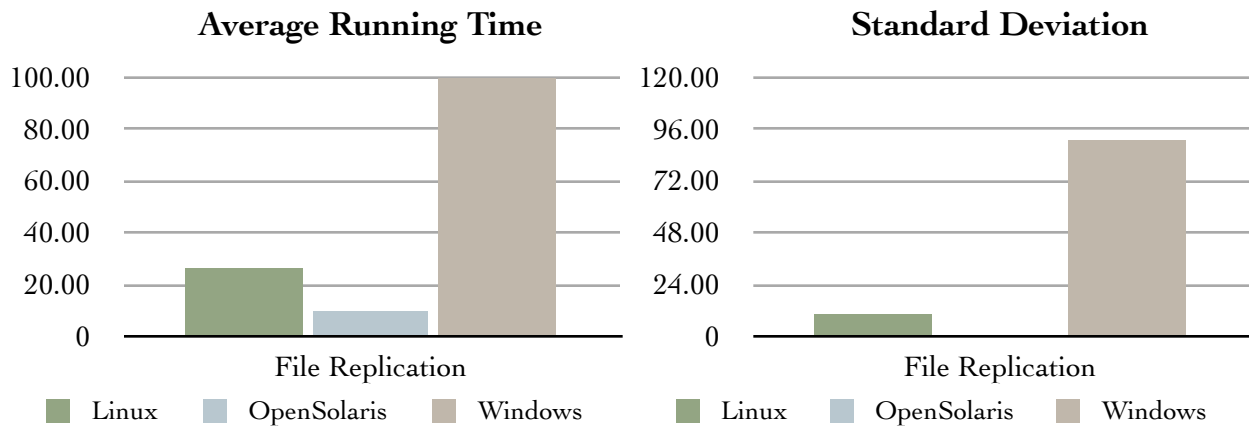


Chart 26. *Stress - C.* Average running time for I/O operations.

Chart 27. *Stress - C.* Standard deviation for I/O operations.

Stress - Java

Our last experiment also puts the machine and the operating systems under heavy stress with multiple concurrent threads. However, the program was written in Java, which means that we have the very same source code for all three operating systems. Not only the operating system will be put under a condition of stress managing its finite resources, but also the JVM (Java Virtual Machine). We have two flavors of JVM: The reference implementation, provided by Sun (now Oracle), running on top of OpenSolaris and Windows, and the OpenJDK running on top of Linux.

Number of Iterations	Concurrent Threads	Total Running Time (milliseconds)		
		Linux	OpenSolaris	Windows
100	103	770,474	937,646	981,864

Table 26. *Stress - Java.* Number of threads and total running time.

As with the case of running the experiment with the C programs putting the operating systems under stress, the equivalent experiment in Java also took a considerable larger amount of time to finish when compared to the other Java-based experiments. Once more this was expected due to having the overhead operations of switching among several concurrent threads.

Time Measurements					
		Iteration Time	Counter Increment	Equation Calculation	File Replication
Average	Linux	7,704.21	6,041.46	7,702.28	314.65
	OpenSolaris	9,375.92	6,703.07	9,374.33	1,507.04
	Windows	9,817.08	6,844.03	9,814.46	4,290.43
Standard Deviation	Linux	393.94	164.11	394.20	333.88
	OpenSolaris	414.03	92.21	414.01	1,453.89
	Windows	1,426.16	178.58	1,426.92	1,884.30
Maximum	Linux	8,673.00	6,533.00	8,672.00	2,644.00
	OpenSolaris	10,399.00	7,302.00	10,397.00	5,923.00
	Windows	13,150.00	7,161.00	13,150.00	7,113.00
Minimum	Linux	6,905.00	5,595.00	6,902.00	67.00
	OpenSolaris	8,478.00	6,490.00	8,476.00	59.00
	Windows	7,878.00	6,302.00	7,878.00	514.00
Margin of Error	Linux	101.48	42.27	101.54	86.01
	OpenSolaris	106.65	23.75	106.65	374.52
	Windows	367.38	46.00	367.58	485.40

Table 24. Stress - Java. Summary Statistics.

Linux performed better than the other two operating systems in this computational stressful environment in all three categories being measured. Again the introduction of semaphores in the operations with floating point and math had a significant impact on the execution times.

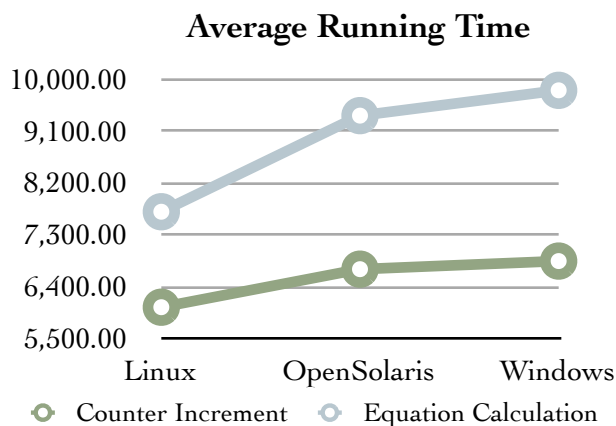


Chart 28. Stress - Java. Average Running Time for operations with integers and floating point and

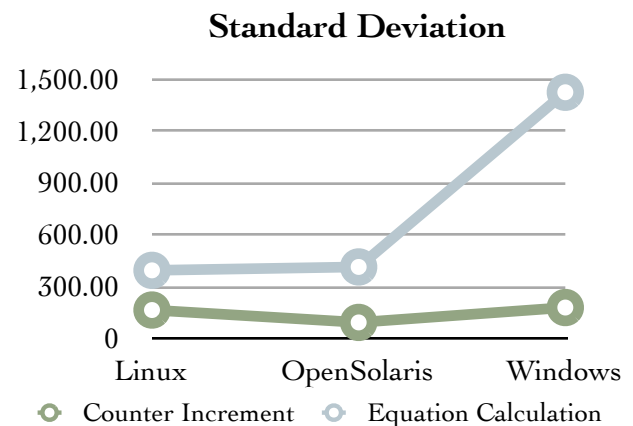


Chart 29. Stress - Java. Standard Deviation for operations with integers and floating point and math.

Chart 30 shows us how each operating system behaved running those operations.

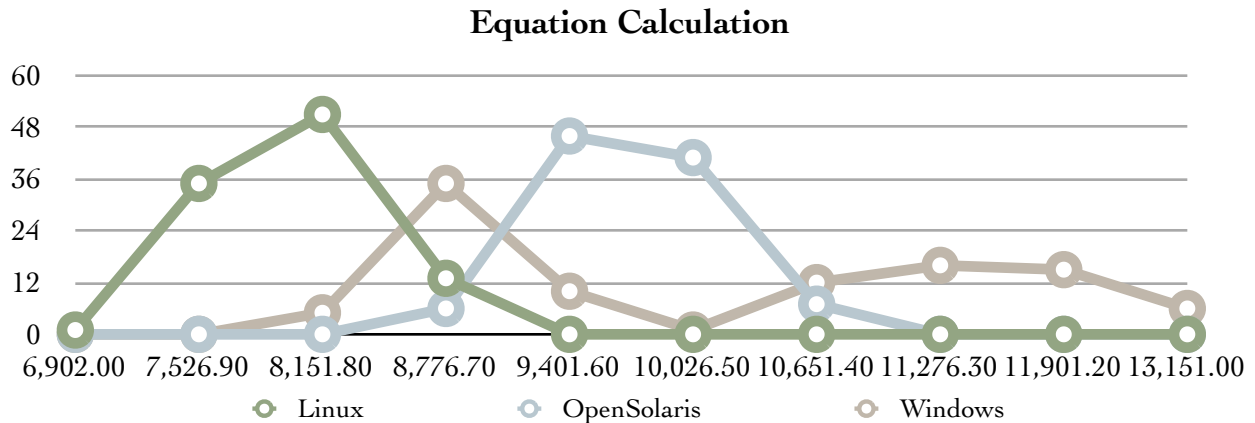


Chart 30. *Stress - Java*. Histogram for operations with floating point and math.

When we compare OpenSolaris and Windows we see that sometimes Windows ran faster than OpenSolaris, while other times it ran slower. Table 25 contains the data of the histogram plotted on Chart 30.

Equation Calculation				
Bin Index	Time Ranges	Linux Frequency	OpenSolaris Frequency	Windows Frequency
1	6,902.00	1	0	0
2	7,526.90	35	0	0
3	8,151.80	51	0	5
4	8,776.70	13	6	35
5	9,401.60	0	46	10
6	10,026.50	0	41	1
7	10,651.40	0	7	12
8	11,276.30	0	0	16
9	11,901.20	0	0	15
10	13,151.00	0	0	6

Table 25. *Stress - Java*. Frequency analysis of the running times for operations with floating point and math for Linux, OpenSolaris, and Windows.

If we analyze the sample data we will find that during the majority of times, OpenSolaris ran faster than Windows.

Equation Calculation			
Time Ranges	OpenSolaris Faster than Windows	OpenSolaris Equal to Windows	OpenSolaris Slower than Windows
6,902.00	0	0	0
7,526.90	0	0	0
8,151.80	0	0	5
8,776.70	0	6	29
9,401.60	36	10	0
10,026.50	40	1	0
10,651.40	5	7	0
11,276.30	16	0	0
11,901.20	15	0	0
13,151.00	6	0	0
Totals	118	24	34
	67.05%	13.64%	19.32%

Table 26. *Stress - Java.* Performance comparison between OpenSolaris and Windows on operations with floating point and math.

Furthermore, when we build the time comparison table, we can see that about 49% of the time Windows took more than 10,000 milliseconds to finish the operations.

Time Range Comparison - Equation Calculation			
Range	Linux	OpenSolaris	Windows
<= 8,151.80	87%	0%	5%
> 8,151.80 and <= 10,026.50	13%	93%	46%
> 10,026.50	0%	7%	49%

Table 27. *Stress - Java.* Time range comparison for operations with floating point and math.

Last, but not least, we discovered that Linux outperformed OpenSolaris and Windows in I/O operations. Its average running time and volatility was considerably smaller than the other two operating systems, as we can see on Charts 31 and 32.

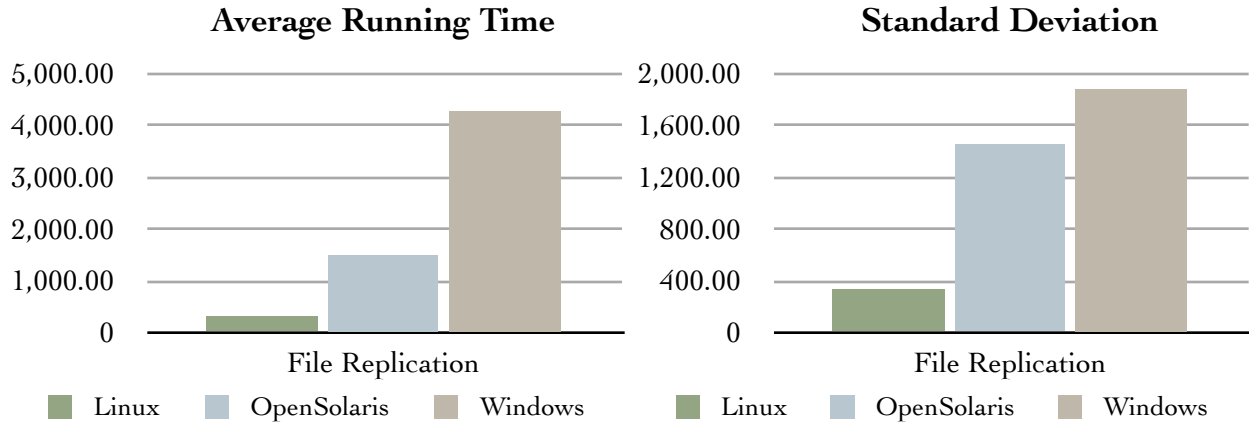


Chart 31. *Stress - Java.* Average running time for I/O operations.

Chart 32. *Stress - Java.* Standard deviation for I/O operations.

The Economics of Operating Systems

The TCO (Total Cost of Ownership) of an operating system is usually hard to be determined. There are many tangible and intangible costs involved (e.g., procurement, installation, maintenance, electricity, etc.). Based on the behaviors measured on these experiments, we can try to estimate with good accuracy some of the most important factors (i.e., response time and electricity consumption) throughout the course of a year.

Regardless of the operating system of choice, there is going to be a physical space, network equipment, insurance, and other equipments and costs that are part of the TCO. That will be exactly the same no matter which operating system is running on the computer. Hence what we can estimate is the marginal cost involved in choosing one operating system versus another. However, these marginal costs are among the most expensive and important costs: electricity consumption, the average transaction time, and the cost per transaction.

One important cost is that of procuring licenses. However, for the sake of this estimate, let's assume that the cost of procuring a Windows license is offset by the cost of having a technician installing Linux or OpenSolaris on a computer, since those are not widely available to be procured pre-installed (nonetheless there are some options available out there).

Our estimate is going to be calculated by averaging the average running time per cycle for all the experiments and projecting the total running time over a period of a year for the operating system with the highest average running time. Then we calculate how much time (in days) the

other two operating systems would require to complete the same task and the electricity cost involved for each one of them.

$$\text{Total Electricity Cost} = \text{Cost per Killowatt-Hour} \times \frac{\text{Watts} \times \text{Hours Running}}{1,000}$$

The average cost of electricity for a commercial establishment in New York was US\$ 0.1536 per hour in December of 2009⁶. A modern server consumes somewhere between 200W and 400W, thus we can safely assume 300W for this estimate.

Averaging the Average Running Times per iteration			
	Linux	OpenSolaris	Windows
Linear C	43.86	39.74	461.43
Threads C	33.36	23.85	421.21
Stress C	1,696.57	3,884.43	2,309.45
Linear Java	2,570.45	2,455.04	2,794.69
Threads Java	2,444.71	2,317.48	2,448.28
Stress Java	7,704.21	9,375.92	9,817.08
Average	2,415.53	3,016.08	3,042.02

Table 28. All Experiments. Averaging the average running time of each of the experiments.

Now we need to calculate how many iterations each operating system would execute in one year. In order to do that we first need to find out the number of milliseconds in a year: 365 days x 24 hours x 60 minutes x 60 seconds x 1,000 milliseconds = 31,536,000,000 milliseconds.

Number of milliseconds in a year	31,536,000,000		
	Linux	OpenSolaris	Windows
Average running time per iteration	2,415.53	3,016.08	3,042.02
Number of iterations per year	13,055,531	10,455,963	10,366,784

Table 29. Calculation of the average running time per iteration and projecting the number of iterations executed in one year's time.

⁶ According to the U.S. Energy Information Administration - http://www.eia.doe.gov/cneaf/electricity/epm/table5_6_a.html

With this data in hand we can calculate our projections. The operating system with the highest average running time per iteration was Windows (3,042.02), hence it becomes our reference point. The total electricity cost to run Windows on a computer over a period of a year (365 days) and execute 10,366,784 iterations is:

$$\text{Total Electricity Cost}_{\text{Windows}} = \$0.1536 \times \frac{300 \times 24 \times 365}{1,000} = \$403.66$$

Now, to execute the same number of iterations as Windows, OpenSolaris and Linux would require, respectively:

$$\text{Time Required}_{\text{OpenSolaris}} = 31,536,000,000 \times \frac{10,366,784}{10,455,963} = 31,267,029,180 \text{ milliseconds}$$

$$\text{Time Required}_{\text{OpenSolaris}} \approx 362 \text{ days}$$

$$\text{Time Required}_{\text{Linux}} = 31,536,000,000 \times \frac{10,366,784}{13,055,531} = 25,041,256,478 \text{ milliseconds}$$

$$\text{Time Required}_{\text{Linux}} \approx 290 \text{ days}$$

Therefore, the total electricity cost to execute the same number of iterations would be:

$$\text{Total Electricity Cost}_{\text{OpenSolaris}} = \$0.1536 \times \frac{300 \times 24 \times 362}{1,000} = \$400.22$$

$$\text{Total Electricity Cost}_{\text{Linux}} = \$0.1536 \times \frac{300 \times 24 \times 290}{1,000} = \$320.53$$

Now we can calculate the cost per iteration for each of the operating systems:

$$\text{Cost per Iteration}_{\text{Windows}} = \frac{\$403.66}{10,366,784} = \$0.0000389379$$

$$\text{Cost per Iteration}_{\text{OpenSolaris}} = \frac{\$400.22}{10,455,963} = \$0.0000382765$$

$$\text{Cost per Iteration}_{\text{Linux}} = \frac{\$320.53}{13,055,531} = \$0.0000245511$$

Conclusion

We started this paper talking about the many reasons why one may choose an operating system. We have studied the Posix and Win32/Win64 threading models, their respective capabilities in performing simultaneous tasks, and lastly we have analyzed the economic aspects associated with the transactions.

The statistical evidence collected and analyzed from all the experiments shows us that the Posix threading model (Linux and OpenSolaris) has clear performance and multitasking advantages over the Win32/Win64 threading model. Furthermore, the file systems ZFS and ext4 were much more efficient and used space more efficiently than NTFS.

The statistical evidence collected from the experiments showed us that the cost of ownership is smaller for Posix based operating systems.

Whichever one is your choice for an operating system, for whatever reason, I hope this paper has advanced your understanding about some of the characteristics of the other operating systems available. I also hope that it may have helped you to reinforce your positions or reevaluate them.

Disclaimers

Microsoft, Microsoft Windows, and Microsoft Windows 7 are trademarks or registered trademarks of Microsoft Inc.

OpenSolaris and Java are registered trademarks of Sun Microsystems, Inc.

Oracle is a registered trademarks of Oracle Corporation and/or its affiliates.

Linux is a registered trademark of Linus Torvalds.

Fedora is a registered trademark of Red Hat, Inc.

Other names may be trademarks of their respective owners.

These trademark holders are not affiliated with Dalmo Cirne. These entities do not sponsor or endorse this paper.