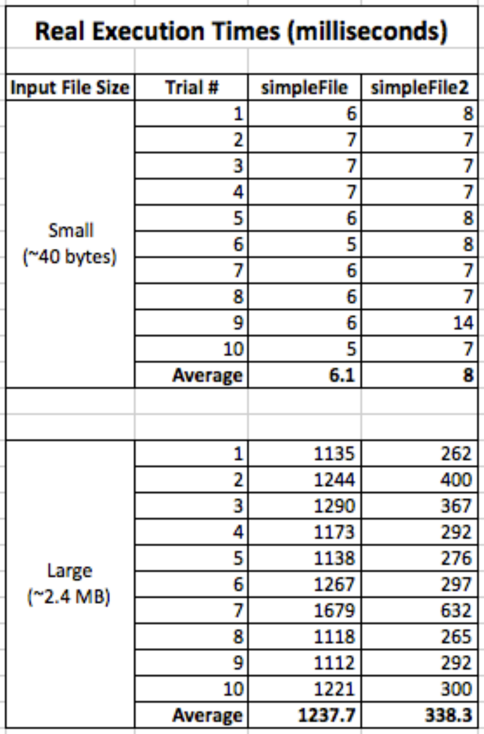
Patrick Sheehan
CSCE 313
Machine Problem 1
1 February 2014

Resources: leepoint.net, piazza.com, connorgarvey.com, mohsin-unaid.blogspot.com

For this project, I wrote a C program that reads a small text file into a buffer, prints the contents to the screen and to a file, and finally deletes the output file it created. I analyzed the system calls when running this program on both a windows and a linux machine. A java program was also written with the same functionality and was run on an android emulator.

I learned about the system calls that correspond to these high level languages' functions. Windows provides many more system functions and each is very specific to its operation. Posix commands on the other hand are much more generalized, and the specific operations are determined by the parameters. For example, Windows has separate functions to create and open a file, whereas linux uses the open command for both. If a file is non-existent when the open command is invoked, a new file is created. When looking at a windows trace, it is much more readable and it is easy to understand what is going on. A posix trace looks very repetitive, with several calls to mainly three or four different functions, and is a little bit harder to understand.

The aesthetic comparison of the above mentioned difference is almost irrelevant. When comparing the timing of a program written with C functions versus one written with primarily system calls, we see that the latter runs much more efficiently. Programming with system calls eliminates a lot of redundancy in the compiled executable. The results are shown below (Figure 1) for a small (~40 bytes) and a large (~2.4 mb) input file, when run with programs using high-level C functions and with mainly posix system calls. For a very small file, the difference is negligible; the C functions are actually faster (by 2ms on average). But for the large file, the system calls performed the task three times faster.



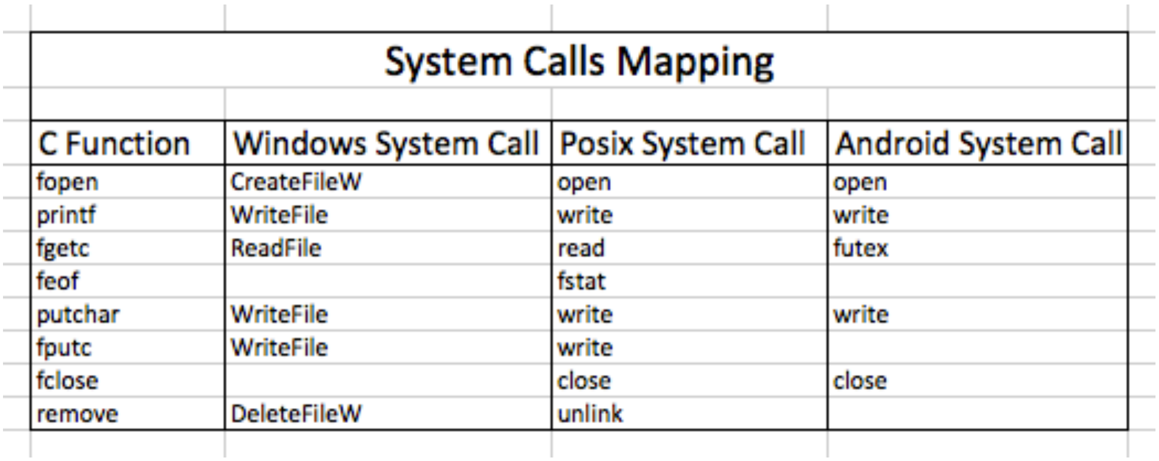
Real Execution Times (milliseconds)

Input File Size	Trial #	simpleFile	simpleFile2
Small (~40 bytes)	1	6	8
	2	7	7
	3	7	7
	4	7	7
	5	6	8
	6	5	8
	7	6	7
	8	6	7
	9	6	14
	10	5	7
	Average	**6.1**	**8**
Large (~2.4 MB)	1	1135	262
	2	1244	400
	3	1290	367
	4	1173	292
	5	1138	276
	6	1267	297
	7	1679	632
	8	1118	265
	9	1112	292
	10	1221	300
	Average	**1237.7**	**338.3**

Figure 1 - Execution Times

Finally, a program with similar functionality was implemented in java for an Android device. Instead of writing to standard output, the text was written to a TextView object (in the applications GUI). The difference for this was that many external libraries are used and many processes were being executed for a simple program (especially since it was on an emulator). However, when analyzing the system calls exclusively for my application, I found that much of the same posix commands were used for the file I/O. Read, write, and open were all used, but futex was used much more prominently. This function waits for the value at a certain address to change. Most likely, a single address was being observed, and each time the loop executed and read a character from the file, it stored it in that address. Then, the character was written to the output file and appended to the TextView of the application.

I learned a lot about the underlying functionality of operating systems from this lab. More specifically, I understand why system programming is advantageous. Although it is more difficult, the increase in efficiency is incredible. The change from C calls to system calls doesn't seem like much, but that's mainly because the task was so simplified. Mathematically, we can see the difference. However the difference between system calls and java calls on a virtual machine was enormous. Admittedly, a lot of time was used to set up the system just to start the program, but it ties into the notion that excessive library usage, while easier on the programmer, hinders performance greatly.



System Calls Mapping

C Function	Windows System Call	Posix System Call	Android System Call
fopen	CreateFileW	open	open
printf	WriteFile	write	write
fgetc	ReadFile	read	futex
feof		fstat	
putchar	WriteFile	write	write
fputc	WriteFile	write	
fclose		close	close
remove	DeleteFileW	unlink	

Figure 2 - System Call Mapping