

Linux Vs FreeBSD: Comparing Kernel Performance

Kirti Chawla

Department of Computer Science
University of Virginia, Charlottesville
kirti@cs.virginia.edu

Taniya Siddiqua

Department of Computer Science
University of Virginia, Charlottesville
taniya@cs.virginia.edu

Abstract

A kernel is the most important component of an operating system. The kernel performance determines the overall performance of an operating system. Most importantly, it defines the superiority (not necessarily the popularity) of an operating system over the other operating systems. Furthermore, measuring the kernel performance empowers us to understand the intricate details of an operating system. This understanding helps us in locating potential design flaws and methods that sap the system performance at runtime. Faulty or inefficient methods for key purposes within the kernel, adds to the existing overhead of the applications. To this effect, we present detailed methodologies to measure the kernel performance of the two widely used operating systems – Linux and FreeBSD. We have performed experiments on one computer, configured to run Linux and FreeBSD as a boot-up option. Our experiments indicate that FreeBSD performs better than Linux. We believe that these results provide one more evaluation parameter for an end-user.

Categories and Subject Descriptors D.4.8 [Performance]: Measurements; H.3.4 [Systems and Software]: Performance Evaluation

General Terms Experimentation, Performance

Keywords Kernel call, context switch, page allocation, file read, kernel performance, Linux, FreeBSD

1. Introduction

It is widely accepted that the performance of an application, among other factors, is determined by the operating system. On the other hand, the performance of an operating system itself is determined by the kernel. This implies that kernel plays a significant role in enabling the performance of the application. Although, we see that large number of operating systems have been developed, yet only a few have been widely accepted by the end-users. Performance (and number) of the applications and operating system is the chief reason behind this phenomenon. In order to objectively determine the role of operating system in the performance of application, we need to isolate portions of application code that uses operating system facilities such as system calls. Through careful and judicious annotations in code, it is reasonably

straightforward to separate the application code into user level code and operating system facilities. Consequently, knowing the cost of operating system facilities can help us determine the impact of operating system on the applications. Herein, it is important to note that, while facilities such as system calls influence the application performance, there are other “hidden” aspects of operating system, which should be taken into account as well. These aspects are context switch time, page allocation latency and more.

To measure the performance of the kernel of an operating system is equivalent to determining the cost of the facilities and “hidden” aspects of the operating system. In order to provide accurate measurement of kernel performance, we must be able to measure the operations related to the kernel with high precision. As we will know shortly, precision in determining execution time of the kernel operations itself manifests as a challenging task. Therefore, careful consideration to select appropriate methods of measurement is required.

In order to understand the role of the kernel performance on the applications, we employ an experimental setup that allows us to differentiate key aspects of the operating system. In our setup, we use one computer (having fixed architecture) with Linux and FreeBSD. We run experiments to determine the execution cost of system calls, context switch time, page allocation latency and file read latency. We measure execution cost through variety of methods with varied precision. We also determine the overhead and resolution of the methods used to present a clear and informative picture of the kernel performance of the two operating systems.

We determine that FreeBSD performs equally or better than Linux on file read operation, page allocation latency and page touch latency measurements. We also find out that kernel call overhead is more in FreeBSD than in Linux. Also, context switch delay is more uniform in FreeBSD than in Linux, while signals perform faster on Linux than on FreeBSD.

This paper is organized as follows. Section 2 describes our experimental setup. Section 3 provides detailed insight into the methodology used. Section 4 showcases the experiments we run and their results. We conclude in section 5.

Also, we noted some interesting artifacts during the course of experiments. Although, we are certain that they will appeal immediately, yet for the sake of brevity we only list them (in appendix).

2. System Configuration

Our experiments are performed on one computer with boot-up option to switch between Linux and FreeBSD. We want to investigate the impact of kernel performance on the applications. By performing the experiments comprising of unmodified code, under two different operating systems, we can infer this impact. For this purpose, we have decided to perform the experiments on one machine. The following table illustrates the hardware and software details of our experimental setup:

Hardware		Software	
CPU ^A	1500.096 MHz	Compiler	GCC 4.2.4
Hard Disk	120 GB		
RAM	1 GB (DDR)		
V-RAM	64 MB	OS1 ^L (Linux)	Kernel 2.6.24-19
Cache	256 KB		
Flags	TSC		
Serial Port	2	OS2 ^F (FreeBSD)	Kernel 7.1
Parallel Port	1		
Floppy	1.44 MB, 3.5''		

* ^A = AMD Athlon XP 2400+, ^L = Ubuntu 8.0.4.1, ^F = FreeBSD 7.1

Table 1: System configuration of experimental setup

It should be noted that both Linux and FreeBSD have monolithic kernel. This allows us to safely ignore nature of kernel as a factor in the experiments and the associated results. We have developed the source code of the experiments on Linux. During the generation of the code, we disable all the flags of `gcc`. Although, FreeBSD can run the binary file generated in Linux, yet we generate the code of experiments on FreeBSD from the source files (with all `gcc` flags disabled). This is done to avoid the invocation of compatibility layer to support Linux binary file by FreeBSD kernel at runtime. Hence, this technique allows us to remove any overhead due to compatibility layer.

We ensure that both the operating system boot-up in text mode with minimal services running in the background. This is another way to reduce “noise” in the system, which can prevent precise measurement.

3. Methodology

A consistent set of rules and methods are applied across experiments to get results devoid of any methodology-related error. Our methodology of the experiments is defined as follows:

3.1 Measurement of execution time

To normalize any variation between two executions of an experiment, we average the outcome of 1000 iterations of the run of experiments. Although, we can use variety of methods for measuring the execution time with varied precision, yet this approach allows us to use methods with reasonably low resolution. Also, we collect the execution timing data in such a way that minimizes the effect of background “noise” such as the context switches. The following code is used for the measurement of execution time in our experiments:

```

00 for (i = 0; i < LOOP_COUNT; i++)
01 {
02
03 Start = Time_Start(); /* start timer */
04
05 Operation(); /* operation to be measured */
06
07 End = Time_End(); /* end timer */
08
09 TimeBuffer[i] = End - Start; /* collect elapsed time */
10
11 }

```

Figure 1: Approach for execution time measurement

We use `TimeBuffer` to compute the average after 1000 iterations. The result is then presented to the user at the end of an instance of experiment. This method is repeated across all experiments.

3.2 Measurement of clock precision

It should be noted that precision of measurement methods are important factor in the experiments. This is due to nature of the kernel operations, which have similar execution times as compared to the overhead of measurement methods. We use two methods in our experiments to measure the execution times of the operations. `gettimeofday()` is a library call, which is provided by the operating system. `rdtsc` is an operation, which is provided by the instruction set architecture of the underlying hardware. Both of them have different resolution and overhead values. The following table summarizes the resolution and the overhead values of the both methods:

Timing Method	Resolution (µs)	Overhead (µs)
<code>gettimeofday()</code>	1	0.238
<code>rdtsc</code>	< 1	0.0413

Table 2: System configuration of experimental setup

In our experimental setup, we have measured the overhead of `rdtsc` to be 62 cycles. `rdtsc` reads from a special register called Time Stamp Counter (or TSC), which measures elapsed clock cycles since boot-up. Consequently, it needs to be converted to time using processor operating frequency. We know that the underlying processor can execute the code in out-of-order manner. For using `rdtsc`, instructions should execute in in-order manner (temporarily). Therefore, a serializing instruction such as `cpuid` is used with `rdtsc` to force the in-order completion.

3.3 Processor type

For our experiments, we did not choose modern multi-core processor. This is due to the availability of multiple clocks on such a processor. Although, some

sophisticated techniques can be used to derive precise timing values from the multiple clocks, yet we chose to avoid it and instead used a uni-processor. The obvious advantage of this choice is to remove the complexity of multiple clocks and related methods to achieve precise timings from them.

4. Experiments and Results

We measure the performance of kernel across four operations over the two operating systems on one computer. We note down any trend in data, infer key insights and present them. The operations that are used for the measurement of the kernel performance are given as follows:

4.1 Kernel Call

All application use kernel calls at various points during their execution lifetime. The number and the frequency of these kernel calls is a factor to determine the performance of the applications. We believe that the task of performance evaluation of the kernel calls can be eased if the application developer annotates the source code with suitable notations that differentiates between the ordinary operation and the kernel calls. We discuss about this aspect in brief in appendix. In our experiments, we measure the execution time of three kernel calls `getpid` (get process id), `getpgid` (get process group id), and `write` (write 1 byte to `/dev/null`).

If kernel calls are invoked multiple times, kernel caches them. This produces the subsequent results in less time than prior kernel calls. We measure both the cached (invoke couple of times before the actual measurement) and the un-cached (invoke the whole program via a script) execution times of the kernel calls.

The following table highlights the cached and un-cached performance of the kernel calls over the two operating systems:

Operating System	Linux		FreeBSD	
	G	R	G	R
<code>getpid</code> (μ s)	0.0136,	0.0128,	0.9914,	0.2473,
{cached, un-cached}	3.61	3.39	6.09	4.91
<code>getpgid</code> (μ s)	0.1367,	0.1274,	1.0821,	0.3263,
{cached, un-cached}	4.34	4.09	14.12	13.37
<code>write</code> (μ s)	0.2986,	0.2776,	1.3229,	0.5816,
{cached, un-cached}	5.12	3.79	8.54	3.95

* G = `gettimeofday()`, R = `rdtsc`

Table 3: Kernel call performance over two OS

It is evident from the aforementioned table that kernel calls on Linux performs better than FreeBSD. Also, cached kernel calls performed better than un-cached kernel calls (as predicted).

4.2 Context Switch

Context switch is a process of saving and restoring processor state so that multiple processes can share a single processor. It is resource intensive operation and

indicative of kernel performance. An optimized kernel would perform context switch in less time than an un-optimized kernel. The interprocess communication (or IPC) provides an elegant mechanism to measure the context switch time of the operating system. In our experiments, we use two IPC mechanisms to determine the context switch time and the IPC overhead over the two operating systems. These mechanisms are defined as follows:

- **Context switch measurement using signals**

While determining context switch time, we need to separate the overhead of the signals. This is done by sending signal within the same process, without invoking a child. The signal handler, which is registered at the onset of the program reverts back to the main function on receiving the signal. We use `raise` to send signal within the single process. To measure the context switch time, we spawn a child process within a parent process and send the two signals between parent-child (`SIGUSR1`) and child-parent (`SIGUSR2`) using `kill`. For proper functioning of this mechanism, synchronization using `sleep` is required between parent and child. We measure the time required to exchange the signal between the parent and the child.

- **Context switch measurement using pipes**

While determining context switch time, we need to separate the overhead of the pipes. This is done by sending 1-word message within the same process, without invoking a child. One pipe is created at the onset of the program. We write on the one descriptor and read from the other descriptor of pipe. To measure the context switch using, we spawn a child process within a parent process and exchange 1-word message between parent-child and child-parent. For this purpose, we create two pipes and close one of the descriptors of both the pipes (within parent and child respectively). We measure the time required to exchange the 1-word message between parent and the child.

The following table illustrates the context switch time and IPC overhead over the two operating systems:

Operating System	Linux		FreeBSD	
	G	R	G	R
<code>signal</code> (μ s)	1.45,	1.43,	2.81,	2.54,
{C, O}	1.33	1.19	2.69	1.64
<code>pipe</code> (μ s)	3.97,	3.6,	2.57,	2.31,
{C, O}	2.19	2.01	2.27	1.34

* G = `gettimeofday()`, R = `rdtsc`, C = Context switch, O = Overhead

Table 4: Context switch latency

It can be inferred from the aforementioned table that on Linux signals perform better than pipes. The opposite is true on FreeBSD. Also, it can be inferred that context switch latency is more uniform on FreeBSD than on Linux.

4.3 Page Allocation

Page fault is an interrupt that occurs when the application accesses a page that is mapped in the address space but not loaded into physical memory. Handling a page fault involves certain latency that has an effect on the response and the execution progress of the application. The latency is determined by the manner in which kernel services the page faults. Therefore, page allocation latency serves as a metric to measure the performance of the kernel. In order to measure kernel using this metric, we construct an experiment that allows us to allocate free pages from memory and fill it with zero values. This page is called zero-filled page.

We determine the cost of allocating the memory region required and latency in writing 1-word to each page in the allocated region. The process of writing to each page is called touching the page. We use fixed page size of 4096 bytes. The following illustration provides the details of our method of allocating and touching the pages:

```

00 for (i = 0; i < MEM_COUNT; i++) {
01  /* warm up cache */
02  Dummy = (char *)malloc(PG_SIZE * MEM_COUNT);
03
04  /* allocate memory */
05  Memory[i] = (char *)malloc(NumOfPages[i] * PG_SIZE);
06
07  /* touch 1-word in every page */
08  for (j = 0; j < NumOfPages[i]; j++) {
09
10     for (k = 0; k < WORD_SIZE; k++) {
11
12        Memory[i][j] * PG_SIZE + k = ':'; /* write 1-byte */
13
14     } /* for (k) */
15
16 } /* for (j) */
17
18 } /* for (i) */

```

Figure 2: Approach for page allocation and touch

We measure page allocation latency across six page sizes ranging from 128 bytes to 4096 bytes. The following graph illustrates page allocation latency as a function of number of pages:

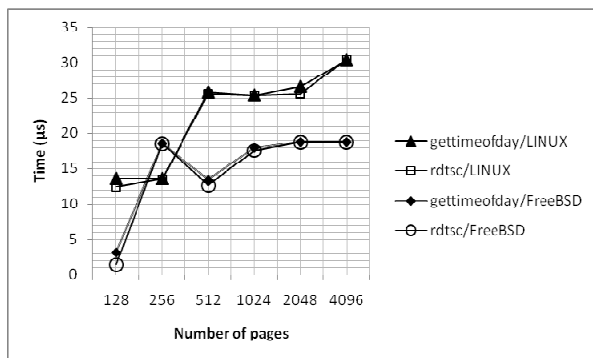


Figure 3: Allocation latency / Number of pages

FreeBSD has lower page allocation latency than Linux except when the number of pages is 256. We attribute this behavior to the way FreeBSD allocate pages to the processes. We measure page touch latency across the six page sizes. The following graph depicts page touch latency as a function of number of pages:

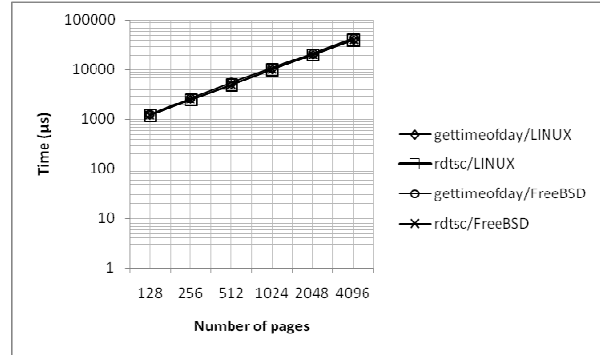


Figure 4: Touch latency / Number of pages

For both the operating systems touch latency increase linearly with number of pages.

4.4 File Read

File read operation is one of most frequently used operation used by applications. Therefore, an efficient mechanism can speed up the application. There are a variety of ways to do file read. We instrument a program that allows us to construct a file of 64 MB filled with random numbers. We vary the buffer-size from 4 bytes to 65536 bytes, during file operations. With this, we understand the role of buffer-size in file read operations. The following describes the two methods for file read operations:

- **read call**

File read operation using `read` call is called un-buffered I/O. Due to the un-buffered nature of the I/O mechanism, there is no support from the operating system or a library to provide optimal buffer mechanism. Therefore, the onus to provide optimal buffer size for file operations, lies on the application. Therefore, in our experiments, we distinguish between the file read latency of cached and un-cached file. This allows us to determine the role of caching on file operation.

- **mmap call**

File read operation using `mmap` call is called memory mapped I/O. This mechanism maps a file on disk to a buffer in memory. It is relatively easy to estimate that the method has slightly higher upfront latency than the `read` call method. In order to simulate un-cached file read behavior, we allocate large swap-buffer having size of equal to physical memory available to us. We then fill it with random values. We do this operation after the file is mapped in the memory. This forces the operating system to swap

out the file buffer. We then measure the latency to move data from memory mapped region to a buffer in memory.

The following illustrations describe behavior of read and mmap operation (cached, un-cached) over the two operating systems:

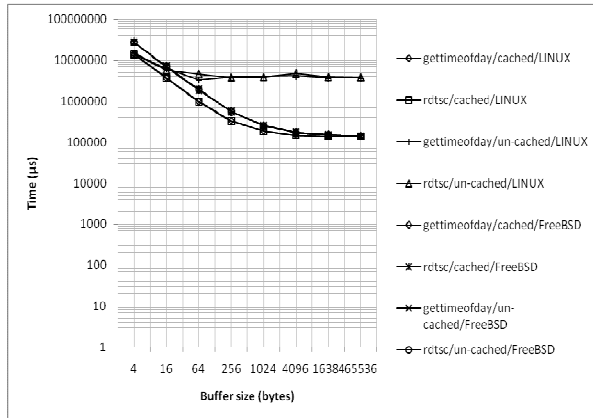


Figure 5: Cached/Un-cached read operation

As seen above, overall Linux and FreeBSD perform in similar manner. Un-cached read operation on Linux has more latency than rest operations over the two operating systems. This suggests that FreeBSD performs equally well in cached and un-cached file read operation.

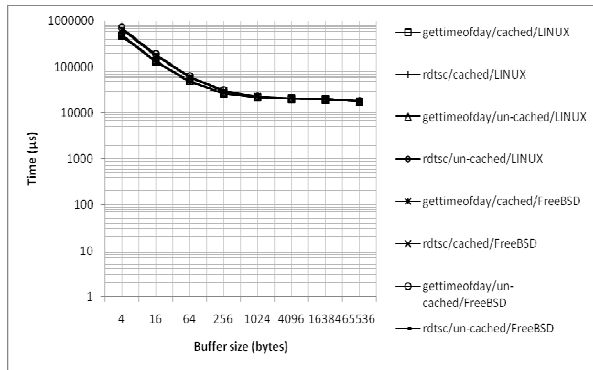


Figure 6: Cached/Un-cached mmap operation

Linux and FreeBSD perform in similar manner for all the version of mmap operation

5. Conclusion

We started with a hypothesis that FreeBSD has better performance than Linux. Our experiments show that even though Linux outperforms FreeBSD in the measurement of kernel call latency, it is FreeBSD that performs better overall.

Appendix

While performing the experiments, we came across an anomalous behavior during debugging of the source code for the experiments. When we used `printf` operation within execution time measurement block, the outcome of measured time was significantly more than when we did not. We attribute this to the overhead of `printf` adding itself to the measured time.

Also, we noted that any given source statement can be distinguished as either a normal or a kernel call. Consequently, a simple scheme to annotate them using the source comments can help differentiate of source code. We provide an illustration of such annotation as follows:

```

00 ...
01 pid = getpid(); /* _SYS_ */
02 ...
03 while (true) { /* _NORM_ */
04 ...

```

Figure 7: Approach for execution time measurement

Here, `_NORM_` and `_SYS_` can be used by an external program to distinguish. Furthermore, this can aid in determining impact of kernel call on application performance (used with kernel call overhead values).

Acknowledgements

We would like to express our sincere gratitude towards our advisor Dr. Marty Humphrey for providing us the opportunity to undertake this intellectually stimulating experiment and for interesting lectures in the operating system graduate course.

1. Reference paper is available at:
pages.cs.wisc.edu/~dusseau/Courses/CS736/CS736-S02/warmup.ps
2. Source code is available at:
www.cs.virginia.edu/~kc5dm/projects/osperf.zip