When Good Kernels Go Bad

(or: Measuring Operating System Overhead)

Mihai Christodorescu

`mihai@cs.wisc.edu`

September 28, 1999

Abstract

I have performed time measurements on stock SunOS 5.6 / Ultra SPARC IIi to gauge the overhead of basic system calls (trivial calls, IPC, I/O, and memory-related). The benchmarking process in itself was a challenge, because from user space there is tremendous indirection and associated overhead to each function call (library or system).

The 64-bit system call *gethrtime()* was found to be useful, with a high precision, very close to the hardware clock precision of 3.3 ns. Several kernel calls were compared to find the kernel call overhead, and *getpid()* proved to be the fastest one. Signals vs. pipes based IPC comparisons clearly presented one-way unnamed pipes as a much better alternative. Further on, the latency of page faults for zero-filled pages revealed some high values, to be reckoned with when allocating large amounts of memory. Finally, I have measured the basic *read()* system call, along with its counterparts *fread()* and *mmap()*, and noticed that *fread()* performs much better than *read()* and that *mmap()* performance recommends it for small read buffer sizes.

The paper concludes by suggesting several ways to improve performance based on the measurements obtained, as well as by a short discussion on how to get relevant measurements. The topic is by no means open to further exploration.

1 Introduction

Measuring the performance of the operating system is of utmost importance. Tuning a piece of software for various purposes can be a difficult task, and impossible to achieve without good measurements to start with. As I intend to show in this paper, getting a meaningful and true value is a tricky process, especially when one deals with the operating system, the basic software layer between the user and its applications and the machine.

As much as possible I have tried to follow good experimental practice when performing measurements, as presented in Miller [2]. In each section, a hypothesis will be formulated, with the variables that come into play clearly specified, results will be summarized, and conclusions will be drawn. In some cases, where this structure is not applicable, it will be pointed out the steps taken to achieve the conclusion. It is significant to mention that I have adopted the arithmetic average as a good approximation of the true time average, according to Jim Smith's suggestions in Smith [1].

All the experiments were executed on the Sun SPARC architecture, with Ultra SPARC II*i* processors clocked at 300 MHz. The machines were running SunOS 5.6 operating system, with stock kernel and libraries. Each machine had 256 MB of main RAM, and were configured with 1 GB of swap space. The machines were connected to an Ethernet network, and were running the network file system AFS. Home directories and other directories were mounted over the network. Most measurements were not affected by network latency, and only in the case of file read times special care had to be taken to obtain values relevant to the local disk system.

The rest of the paper is organized as follows: section 2 presents the various methods available to programmers to measure time in Sun Solaris. Section 3 attempts to measure the lapsed time of a bare-bones kernel call, by timing and comparing several supposedly trivial calls. Section 4 computes the overhead associated with process context switching, while section 5 presents page fault latencies incurred when a new page is allocated to memory. Disk read timings are presented in section 6, all to be concluded in section 7, where I attempt to generalize the experience thus gained, and propose some ways of alleviating the delays.

2 Choosing A Time Machine

The Solaris operating system provides a large number of time-related functions, ranging from specialized libraries (for the C, Fortran, and Tcl/Tk languages), to system calls. Thus, a quick survey of the various ways of getting the time, and measuring time intervals is called for.

The goals of this selection process were to find the most precise benchmark method available on stock OS and hardware, keeping in mind that the desired precision was hardware bound. Specifically, for 300 MHz, the hardware clock precision is $\frac{1}{300\ MHz} \approx 3.3\ ns$. Since I was interested in measuring time as an interval, rather than an absolute value, functions that returned intervals (such as interval timers) were considered for comparison, along with time-of-the-day or time-since-Epoch functions.

A quick lookup[1] turned up the following list of time-related library entries and system calls: *alarm(), clock(), clock_settime(), clock_gettime(), clock_getres(), dsecnd(), ftime(), gethrtime(), gethrvtime(), gettimeofday(), getitimer(), setitimer(), napms(), secnds(), second(), time(), timeout(), timer_settime(), timer_gettime(),*

[1]The `man` command proved to be a great tool, with its `apropos`!

timer_getoverrun(). Unfortunately, only *gethrtime()*, *clock_gettime()*, and *timer_get/settime()* return values in the range of nanoseconds, all the other functions being for more general, less precise purposes and having a specified resolution of microseconds or even seconds. Further study showed that *clock_gettime()* has a resolution of 1000 ns, which pushes it back in the microseconds range for practical usage.

The documentation for *gethrtime()* references the hardware timers available on the Sun SPARC platform. Disassembling the library call revealed it consists of a single system call, a software trap `0x24` (known as `ST_GETHRTIME` in the C library header file `trap.h`). This seems to indicate that the overhead associated with this call is minimal, and that the precision is close to the one provided by the hardware clock of the system. Later tests targeted to measuring the time of a single ADD instruction (1 cycle in execution length on the SPARC architecture, since it is a register-to-register instruction) showed that *gethrtime()* has a resolution very close to the actual CPU clock, with measured interval values in the range 2 - 4 ns. On a 300 MHz processor, this value is almost identical to the average cycle time (3.3 ns), and I therefore selected *gethrtime()* for use in the rest of the experiments.

3 Trivial Kernel Calls

The overhead of a system call is determined by several factors: the need to save the process state, the interrupt processing, the time spent transferring data between the kernel space and the user space (similar to the setup time a new subroutine call in an user program). In order to measure this overhead as accurately as possible, without modifying the kernel in any way, the overall time of a system call is obtained, and the main problem becomes selecting the least expensive system call.

The system calls best-suited for this purpose are the ones that return a fixed value, maybe something that is accessible in the registers already (so they do not incur a trip through the memory hierarchy), or that do not involve any computation or indirection. Obviously, I/O bound calls are excluded from start, as well as calls that involve other processes, and which might generate a context switch[2]. The following calls were selected for testing: *getpid()*, *getppid()*, and *uname()*. The first two return information, that although volatile when related to total system uptime, is constant with respect to the running process, while *uname* provides data that was set once at boot time (or hard-coded at compile time).

[2]The context switch overhead is related to the system call overhead, but has its own quirks, and it is treated in section 4.

System call	Avg.* exec. time (ns)
getpid()	1,888.95
getppid()	1,867.89
uname()	3,980.82

*Averaged over 1,000,000 iterations.

The above values were adjusted for the overhead incurred by the *gethrtime()*. For example, if the measurement is performed as shown here:

```
start_time = gethrtime();
<instruction to be measured>
end_time = gethrtime();
elapsed_time = end_time - start_time;
```

Then the `elapsed_time` should be adjusted by an `overhead_offset` computed as follows:

```
start_time = gethrtime();
end_time = gethrtime();
overhead_offset = end_time - start_time;
```

All the measured values presented in this paper are thus adjusted.

For comparison, the not-so-trivial system call *times()* was measured, producing the average value of 9,667.44 ns. The large differences are mostly accounted by the extra work performed inside the body of the call (the actual utility) by the kernel: for *times()* it involves reading the system clock and converting the value into clock ticks since a certain pre-established (*read* stored) checkpoint in the past. For *uname* the extra work involves putting together the information about the system into the various memory locations that compose the `struct` argument. The similar times of *getpid()* and *getppid()* reflect the commonalities they share (they are variations of the same functionality (retrieving a process ID), and imply that their work involves just reading some value from the kernel memory (very fast, even if not cached already).

4 Context Switches

The kernel switches among processes in several circumstances (when *yield()* is called, or the currently running process blocks on an I/O calls, or the time quantum of the currently running process runs out). The scheduler then decides which is the next process to execute, but the majority of the time is spent saving one process' state and restoring the incoming process' state. Optimizing a program around such behavior can lead to better multitasking properties.

Since instrumenting the kernel and the system libraries was not an option, the only way to measure a context switch delay is to force one, thus deoptimizing the process scheduling policy, but achieving some fairly accurate results. One way to force a context switch is to use pipes between two processes and blocking reads

/ writes to make the two processes wait for each other. Another way is to use the standard signal semantics between two processes, to force immediate delivery and return of signals. Each of the two methods has its advantages and disadvantages, which affect their performance quite dramatically, and both their results are illustrated below.

IPC method	Average* round-trip time (ns)	Estimated† context-switch time (ns)
signals (*kill()*, *signal()*)	85,644.9	42,822.45
pipes (*pipe()*, *read()/write()*	44,505.5	22,252.75

*Averaged over 100,000 iterations.

†Estimated as $\frac{1}{2}$ of the round-trip time.

Why the big difference? Both methods involve two context switches, but signals also mean spending more time in the kernel space to modify the process pending signal queues. Signals bring in overhead due to the invocation of the signal handler, which, although it executes on the same stack as the signaled program, has to save and restore the program state. Combine this with the fact that pipes use their own memory space to transfer the data, and one can clearly see that signals are not at all an optimal inter-process communication method.

5 Page Faults

Since the advent of virtual memories, page faults have been looked down upon as an evil operating systems cannot do without. Usually page faults refer to the fact that certain process memory areas might be swapped out to disk, and when accessed they need to be brought in, with all the overhead of the I/O subsystem. In this section I will concentrate on a different kind of page faults, the ones generated by newly allocated memory pages.

When the operating system is required to allocate memory, it assigns a virtual page to the process and it associates with it a physical page (a.k.a. page frame). This is a fairly fast procedure. The delay turns out to occur when the process actually "touches" that page (either by reading or writing it). The UN*X semantics require that freshly allocated page frames be cleared (filled with zeroes) in their whole entirety. This uses precious system time. The measurements below are meant to shed some light on how this requirement can affect system performance.

For this test, a program was written to allocate a large chunk of memory (8 MB) and access each page once, enough to trigger the "clearing" process of the operating system. The total time was then divided by the number of pages to obtain an average page fault initialization time. The memory was allocated on a page boundary, SunOS 5.6 using a page size of 8 k.

Process	Avg.* Latency (ns)
Page fault for zero-filled pages	73,136.77

*Averaged over 1,024,000 iterations.

Intuition indicates that this kind of overhead occurs more often on a lightly loaded system, or in the beginning stages of boot and uptime of an operating system, until the memory "warms up." Programs that are performance- or timing-bound (e.g. real-time applications) need to find different policies for allocating large amounts of memory: either do not allocate it all at once, or do so in once batch at start-up time, not while running.

There is a question of whether such functionality is really needed from the operating system. There is a compatibility issue to be approached here: certain program and libraries might rely on freshly allocated memory to be zeroed for their correct operation. Although not documented, the use of this operating system side-effects creates dependencies that prevent efficiency updates in the kernel code or the system libraries.

6 File Reads

Finally, the performance of the I/O subsystem was tested, in order to get a comparative overview of the various methods of reading from a file. The chosen methods were: *read()* (unbuffered, synchronized read from stream), *fread()* (buffered read from stream), and *mmap()* (memory mapped reading from stream). They were tested on an 128 MB file (completely mmap-ed to memory for the third case), with read sizes ranging from 8 to 65536 bytes.

The SunOS operating system automatically buffers any files that are read, thus making consequent test irrelevant (they become reads from the memory/cache hierarchy). In order to prevent such effects, the following procedure was used to clear up the buffers: the program allocated a memory area the size of all the physical memory installed, and wrote it with bogus values in its entirety (a process not dissimilar to the one performed by the OS for zero-filled pages), then freed it. In the process, the OS was forced to dump all the buffers, thus making sure that any subsequent file accesses would be from the disk, not from the cache.

As expected, *fread()* performed much better (see figure 1) than *read()* (keep in minding that the two graphs are drawn using logarithmic scales). *mmap()* seems to perform better for small buffer sizes (see figure 2), only to have its times worsen as the read buffer size increases. This indicated *mmap()* is implemented tightly coupled to the system page size (8192 bytes on SunOS / SPARC), which makes getting data which spans several pages a slow process (due to the fact it generates page faults).

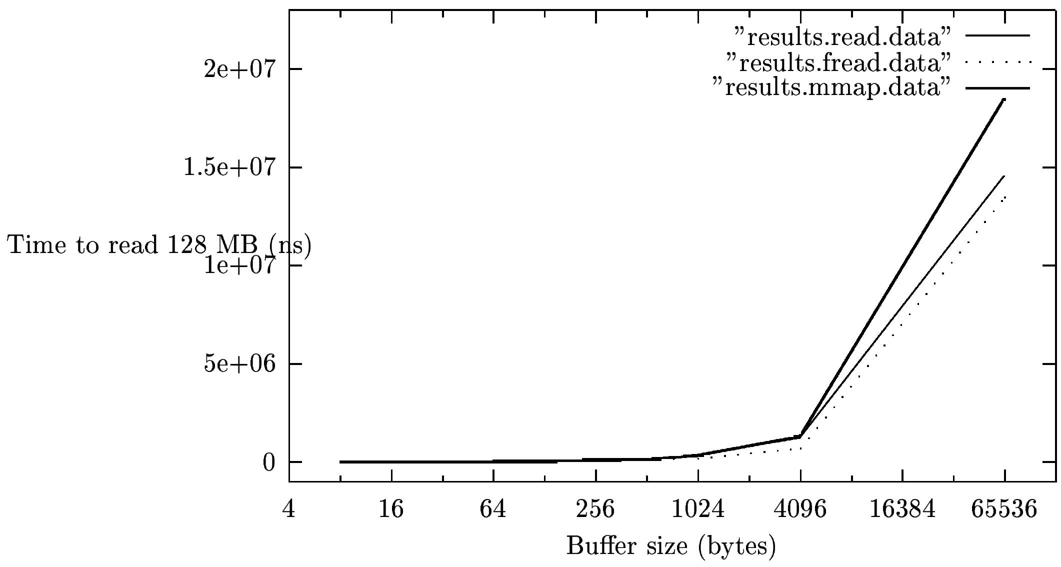

Figure 1: Large buffer sizes make *fread()* a winner. (x in log_2 scale)

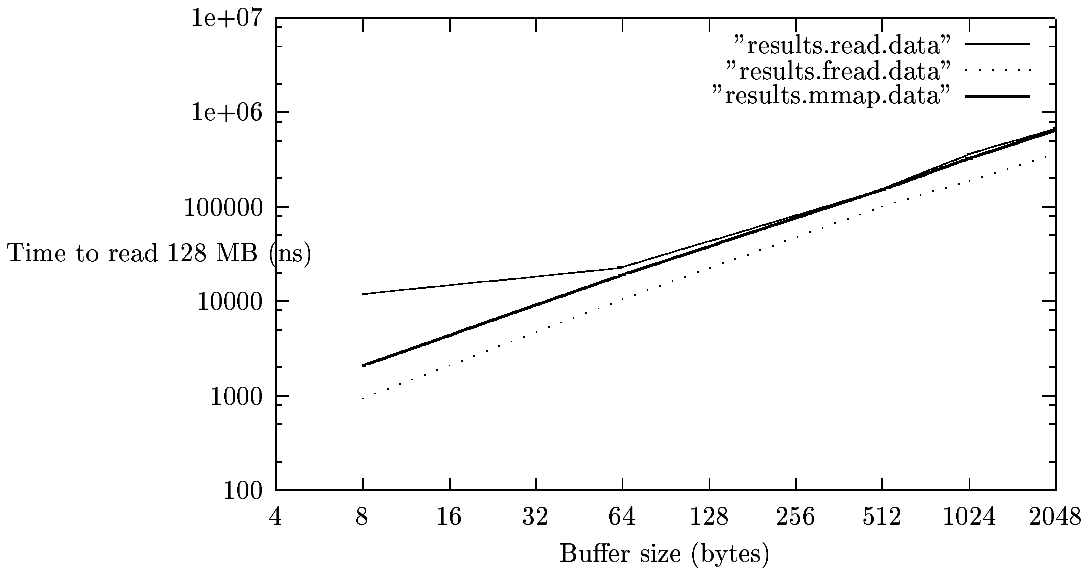

Figure 2: Zoom in for small buffer sizes shows *mmap* a winner. (x in log_2 scale, y in log_{10} scale)

7 Conclusions

Operating system performance cannot be captured in a bunch of numbers. Its response time and throughput are highly dependent on the environment (for example, my measurements were affected from time to time by the Condor job scheduler running in the background, as well as by the NTP daemon continuously adjusting the system clock, thus creating clock skew). I have tried to make the measurements as unbiased as possible, by staying on the same machine as long as possible, and averaging the results over a lot of iterations. Initially I started with a low number of iterations and then increased them until the values stabilized, and the average value was consistent. This fine-tuning process was performed for each experiment.

When using the facilities offered by the operating system (by the means of system calls), the software developer has to take into account their performance. Signals are a much worse alternative to pipes when it comes to continued and prolonged interprocess communications. Also, each kernel call incurs some overhead, which adds up to large numbers if used frequently.

It is worth noticing that the allocation of large areas of memory is affected by operating system semantics; a good solution would be to have a separate thread or process "initialize" the newly allocated memory areas by accessing them before they are actually used or even actually needed. On a somewhat similar note of usage, buffered I/O fares much better than unbuffered output, and sometimes better than memory mapped files.

The source code of the test programs can be found at `http://www.cs.wisc.edu/~mihai/my_work/projects/os_measurements/os_measurements_uwcs736f1999.tar.gz` (4,082 bytes).

References

[1] J. E. Smith: *Characterizing computer performance with a single number*, Communications of the ACM, Volume 31, No. 10, Oct. 1988

[2] Barton Miller: *Paper Assignment #1*, Computer Science 736, University of Wisconsin, Madison, Fall 1999