PERFORMANCE MEASUREMENT

Akanksha Singh (asingh27)
Shashank Jha (sjha5)

INTRODUCTION

GOAL

For this project, we have to measure the time of performing various operations in an operating system. This will help us get an idea of performance of various operations, which in turn will help us understand how an operating system works internally. Keeping these performance measurements in mind will help us design better software which is the ultimate goal of this project.

LIST OF EXPERIMENTS

S. No.	**Name of Experiment**	**Performed By**
1.	Measuring Time Overhead	Shashank Jha
2.	Loop Overhead	Akanksha Singh
3.	Procedure Call Overhead	Akanksha Singh
4.	System Call Overhead	Akanksha Singh
5.	Task Creation Time	Akanksha Singh
6.	Context Switch Time	Shashank Jha
7.	RAM Access Time	Shashank Jha
8.	RAM Bandwidth	Shashank Jha
9.	Page Fault Service Time	Akanksha Singh

WORKLOAD

Hardware specifications - 2 days (4 hours)
CPU, Scheduling, and OS Services - 10 days (25 hours)
Memory - 15 days (25 hours)
Report - 2 days (6 hours)

MACHINE DESCRIPTION

1. Processor
    Model- Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz
    Cache sizes:
        L1 Data - 2 X 32 KB
        L1 Instruction - 2 X 32 KB
        L2 - 2 X 256 KB
        L3 - 3 MB
2. Memory bus: 100 MHZ
4. RAM size: 12 GB (2014MB on VM)
5. Disk:
        Capacity - 1000 GB
        RPM - 5400

References:
http://www.cpu-world.com/CPUs/Core_i5/Intel-Core%20i5-6200U%20Mobile%20processor.htm
cat /proc/cpuinfo
http://www.bestbuy.com/site/hp-envy-x360-2-in-1-15-6-touch-screen-laptop-intel-core-i5-12gb-memory-1tb-hard-drive-hp-finish-in-natural-silver/5238300.p?skuId=5238300
Command: wmic cpu list full

CONSTRAINTS DUE TO USE OF VM

To conduct our experiments, we installed Ubuntu 16.04 in Oracle VM VirtualBox. Hence, only one CPU core was used. Similarly, all 12 GB of RAM could not be used. We use the following description for analysis of our experiment results.

Updated machine description:

1. Cache Size:
   L1 Data - 1 X 32 KB
   L1 Instruction - 1 X 32 KB
   L2 - 1 X 256 KB
2. RAM Size: 2016 MB
3. Disk: 15 GB
4. Operating system (including version/release) - Ubuntu 16.04.1
5. Language- C
6. Compiler- gcc (Ubuntu 5.4.0-6ubuntu1~16.04.2) 5.4.0 20160609

We believe the use of virtual machine will lead to much higher cycle counts for all operations as compared to a physical machine as the virtual machine has constrained resources and additional overhead. However, the relative values and general trends in observations should be maintained.

EXPERIMENTS

I. <u>CPU, Scheduling, and OS Services</u>

1. Measuring Time Overhead

<u>Methodology</u>

We measure the processor cycles using function getCycles() consecutively twice and subtract the values to get the number of cycles needed to measure time. We then repeat this 50000 times and average the results to get average number of cycles and average time for measuring time.

Results

Base Hardware Performance (ns)	Software Overhead Estimate (ns)	Total Estimated Time (ns)	Measured Time (ns)
15	15	30	904

Results Discussion

As frequency of our machine is 2.4 GHz, it takes approximately 0.5 ns for each processor cycle. Based on the functioning of rdtsc, we estimate that the hardware takes about 30 cycles to measure time. We guessed that the software adds an additional 15 cycles overhead, making our predicted operation time 30 ns.

However, our measured operation time was drastically higher. Our getCycles() function uses serialization on cpuid which adds extra overhead to rdtsc, thus resulting in higher time for measuring time. However, we chose this method as it gives more accurate processor cycle. Also, as measuring time overhead is being subtracted from all other measurements, this will not affect the accuracy of our experiment results.

Also, since we are performing are experiments on Virtual Machine, additional overhead must be considered.

Reference: How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures

2. Loop Overhead

Methodology

We used a simple for loop that initializes a variable i to 0, increments it by one and checks for condition i<1. As a result the loop runs exactly once. Also, there are no operations within the loop (to avoid additional overhead). We measure cycle count of the processor right before and after the loop and subtract the values to get number of cycles needed for a loop. This operation is performed 50000 times and averaged to get average number of cycles and average time for a loop. Measuring time overhead was subtracted from this to get actual time for a loop. The experiment was performed 10 times and mean and standard deviation across all results were calculated.

Results

Base Hardware Performance (ns)	Software Overhead Estimate (ns)	Total Estimated Time (ns)	Measured Time (ns)
5	30	35	190

Results Discussion

We estimate that the hardware will take about 10 cycles to initialize the counter, check the loop condition, enter the loop and increment the counter giving a base hardware performance of 5 ns. We estimate the software overhead to be around 30 ns. But, our measured loop time stands much higher at 190 ns.

Source: http://www.agner.org/optimize/instruction_tables.pdf

3. Procedure Call Overhead

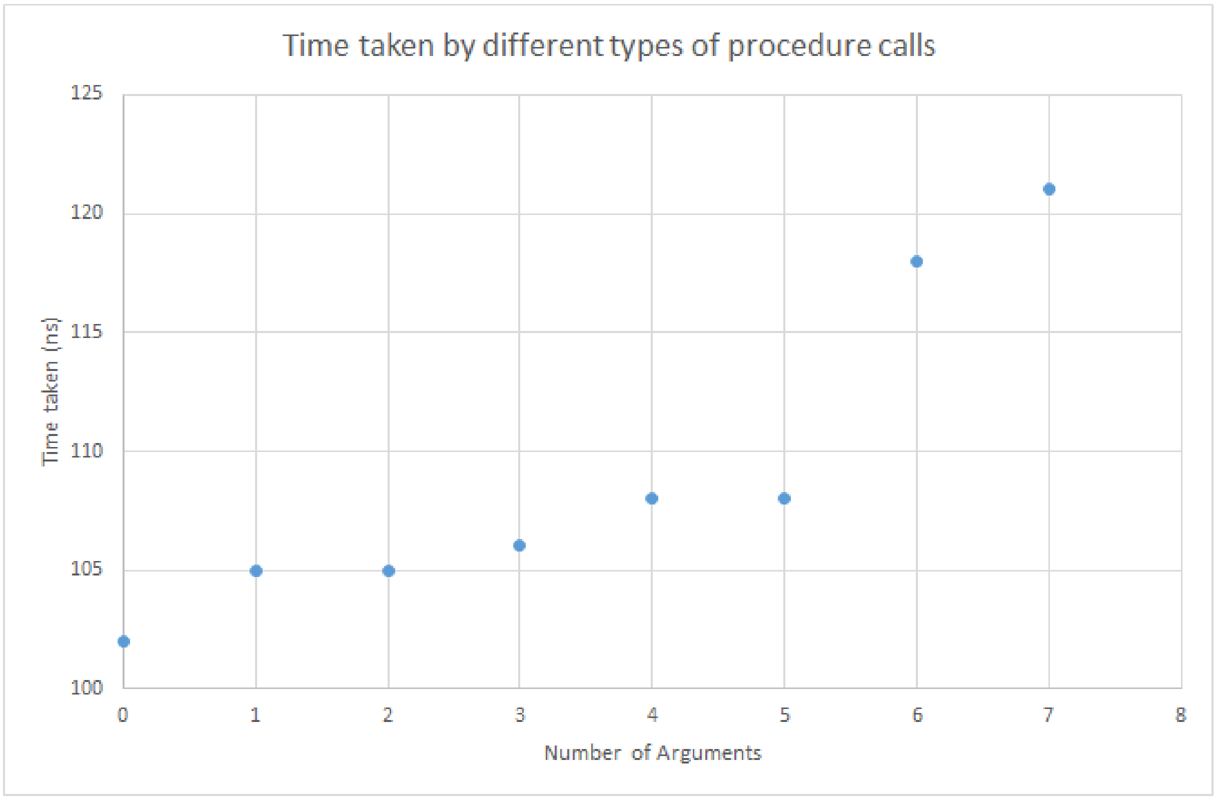
Methodology

We used 8 procedure calls, each taking 0, 1, 2, 3, 4, 5, 6 and 7 integer arguments. Each procedure was called passing randomly generated integers as arguments. Each procedure returns the integer 10 and performs no other operation. This was done so that we could compare time purely as a function of number of arguments, with no additional overhead.We measure cycle count of the processor right before and after each type of procedure call and subtract the values to get number of cycles needed for each type of procedure call. As a result there is no loop overhead. All these operations were performed 50000 times and the results were averaged to get average number of cycles and average time taken for each type of procedure call. Measuring overhead was subtracted to get actual time for each type of procedure call. The experiment was performed 10 times and mean and standard deviation across all results were calculated.

Results

No of Args	H/w Estimate (ns)	S/w Overhead (ns)	Total Estimated Time (ns)	Measured Time (ns)
0	2.5	5	7.5	102
1	5	10	15	105
2	7.5	15	32.5	105
3	10	20	30	106
4	12.5	25	37.5	108
5	15	30	45	108
6	17.5	35	52.5	118
7	20	40	60	121

Results Discussion



We estimated that both hardware and software will take more time to execute a procedure call with more number of arguments as with every increase in number of arguments, there is one more variable that needs to be pushed on to the process stack. On measuring time, we observed increasing values with increase in number of arguments. However, the increase was not as high as we had anticipated. We believe this could be because the data is cached over multiple iterations of a loop.

Increment overhead of an argument:

The increment overhead of an argument was found to be around 2.7 ns.

4. System Call Overhead

Methodology

We used the system call getpid() as it is a minimal system call that returns the process id of the current process. We measure cycle count of the processor right before and after the system call and subtract the values to get number of cycles needed for the system call. As a result there was no loop overhead. This was repeated 50000 times and the results were averaged to get average number of cycles and average time taken for the system call. Measuring overhead was subtracted to get actual time. The experiment was performed 10 times and mean and standard deviation across all results were calculated.

Results

Base Hardware Performance (ns)	Software Overhead Estimate (ns)	Total Estimated Time (ns)	Measured Time (ns)
50	100	150	1645

Results Discussion

We estimate that a system call will take around 150 ns to complete. However, it was observed that system calls take a much higher value at 1645 ns on average. We believe that this is due to the fact that system calls have to interrupt into the kernel which drastically increases a system call time. A system call requires moving from user mode to kernel mode, and then back to user mode. All these factors are most likely responsible for the large time value observed.

Comparison to cost of procedure call

We observe that the cost of a system call is much higher than that of a procedure call. This is due to the fact that system calls have the additional overhead of trapping into the kernel, changing privileges etc.

Source: https://www.ibm.com/developerworks/community/blogs/kevgrig/entry/approximate_overhead_of_system_calls9?lang=en

5. Task Creation Time:

Methodology

a) Process creation time
We use fork() system call to create a new process. We measure the cycle count of the processor right before the call to fork. Once fork is called, we make sure that parent doesn't execute and waits for the completion of child by calling wait() system call. As soon as the child process is created we measure the cycle count of the processor (within the code for child). We then use a pipe to pass the cycle count to the parent and then kill the child. The parent then subtracts both values to get the number of cycles needed to create a new process. This was repeated 1000 times and the results were averaged. Measuring overhead was subtracted to get actual time. There was no loop overhead. The experiment was performed 10 times and mean and standard deviation across all results were calculated.

b)Kernel thread creation time
We create a kernel thread using pthread_create(). We measure the cycle count of the processor right before and after thread creation and subtract the values to get the number of cycles needed to create a thread. The new thread runs a function that prints thread created successfully and returns. The experiment was repeated 1000 times and results were averaged to get average number of cycles and average time for creating a kernel thread. Overhead of measuring time was subtracted. There was no loop overhead. The experiment was performed 10 times and mean and standard deviation across all results were calculated.

<u>Results</u>

Type	H/w Estimate (ms)	S/w Overhead (ms)	Total Estimated Time (ms)	Measured Time (ms)
Process	8	20	28	63.5
Kernel	5	16	21	29.5

<u>Results discussion</u>

We estimate the process creation time to be 28 ms and the kernel thread creation time to be 21 ms. As both processes are intensive system calls, we expected to get considerably higher values. However, the values measured through our experiments was much higher.

<u>Comparison</u>

As predicted, kernel thread creation required less time than process creation. This is most likely due to the fact that threads are considered light weight processes and shares many of the parent process's resources. A new process through fork() on the other hand requires a separate copy of every resource.

6. Context Switch Time

<u>Methodology</u>

a)Process context switch
To check the context switch time between processes, we created two processes using fork() method and two pipes for reading and writing purpose. Initially one of the process writes in the pipe and then moves to next pipe to read. As nothing has been written on next pipe, it has to wait. This forces context switch and now the second process will read from first pipe and writes to second pipe. Again, same process will wait for new bytes to read from first pipe. This again forces context switch. Now, first process will read from second pipe and writes to first pipe. Hence, context switch is forced again. This process is repeated 10000 times. We then measured the total time. A point to note is that the total time is for 20000 context switches as each loop will cause 2 switches.

b)Thread context switch
To check the context switch between threads, we had to create two threads using pthread() method. As with process context switch, we used pipes to forces context switch and repeated the experiment 10000 times. Hence, the total time measured was for 20000 context switches.

<u>Results</u>

Type	H/w Estimate (ms)	S/w Overhead (ms)	Total Estimated Time (ms)	Measured Time (ms)
Process	16	30	46	126
Kernel	8	18	26	92

<u>Results Discussion</u>

We estimate relatively higher values for context switch (compared to other operations) as we predict that during context switch it is possible that a process/ or thread may be switched out into memory. Once again, we estimate kernel thread switching to take much lesser time as the data for thread is relatively less (due to shared resources with parent) and thus needs to be brought in from the cache only. Our relative estimates between thread and process are correct. However, the measured values are much higher than our estimates.

<u>Comparison</u>

Process context switch is more heavy weight as process has a separate copy of all its resources. It therefore must need to maintain a lot more data structures and perform more operations compared to a thread during context switch.

II. Memory

1. RAM Access Time

Methodology

To check the memory latency for different levels of RAM, we made series of integer arrays and then randomly accessed array values. The array’s size varied from 128 bytes to 4 MB which we expected will cover all the hierarchy levels. Each array was accessed 1000 times using rand() function to make sure that all accesses are random in nature.

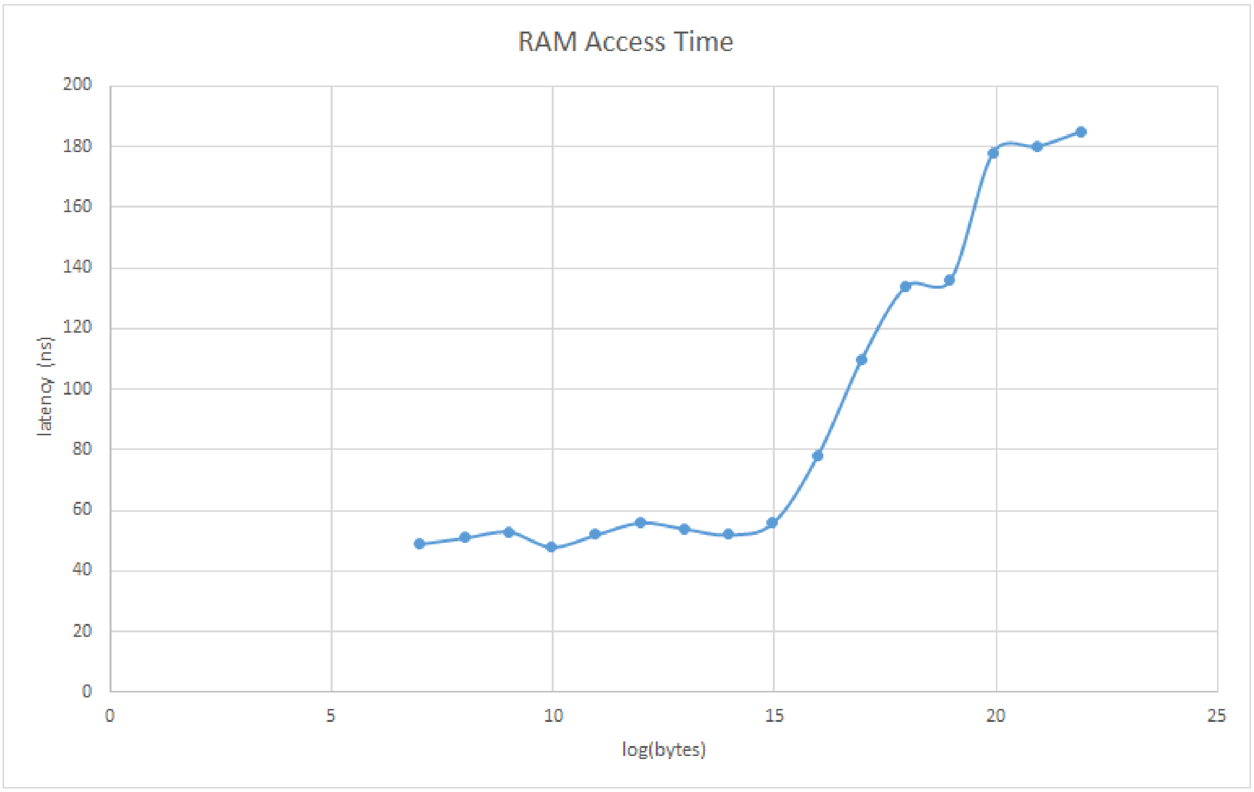
Results

Array Size	H/w Estimate (ns)	S/w Overhead (ns)	Total Estimated Time (ns)	Measured Time (ns)
128 B	15	10	25	49
256 B	15	10	25	51
512 B	15	10	25	53
1 KB	15	10	25	48
2 KB	15	10	25	52
4 KB	15	10	25	56
8 KB	15	10	25	54
16 KB	15	10	25	52
32 KB	15	10	25	56
64 KB	15	10	25	78
128 KB	25	20	45	110
256 KB	25	20	45	134
512 KB	120	100	320	136

1 MB	120	100	320	178
2 MB	120	100	320	180
4 MB	120	100	320	185

Results Discussion

We predicted consistent L1 (25 ns), L2 (45 ns) and memory (320 ns) access times. Moreover, we predicted that L2 access time would be higher than L1 access time and memory access time would be much greater. Our observed values, though higher than predicted values, are consistent for L1. However, beyond L1 we did not observe separate plateaus for L2 and memory. We observed a general increase in memory without clear distinction between L2 and memory. We believe that the use of a VM has caused this uneven increase in L2 and memory access time.



Graph showing RAM access time varying according to array size

Reference: lmbench: Portable tools for performance analysis

2. RAM bandwidth

<u>Methodology</u>

To check RAM bandwidth, we had to measure time taken for both the read and write operations. For reading, we created a 4 MB array and initialized all its value. Then, we read three values which were elements at index i, i/4 and i/2 (where i is value of loop variable.) The values of these elements were added and used later. For writing, the value obtained from reading test above was inserted to array position determined by rand() function. These two processes were done 500 times and average time per read and write were calculated.

<u>Results</u>

Memory (MB)	Operation	H/w Estimate (ns)	S/w Overhead (ns)	Estimated Time (ns)	Measured Time (ns)
4	Read	100	200	300	586
4	Write	100	200	300	1632

<u>Results Discussion</u>

We expected both memory read and write operation to take about the same amount of time. However, it was observed that memory write takes much longer (about 3X) than memory read operation. We predict that the added overhead is either due to large write buffer.

3. Page fault service time

<u>Methodology</u>

We used the mmap function to map standard input (fd =0) to memory location 0x9000000. Moreover, we mapped one page of memory (4KB = 4096 bytes). We then copied the string "This causes a page fault" to the address returned by the mmap function. This copy results in a page fault. We measured the processor cycle right before and after the page fault and subtracted the values to get the number of cycles for page fault. We repeated the experiment 50000 times and averaged the results to get the average time and average number of cycles for a page fault. The experiment was

repeated 10 times and mean and standard deviations across all experiments were calculated.
At times the mmap function was resulting in segmentation fault (core dumped). We assume that this is because we have hardcoded the start address, the address might not always be available and thus leads to the segmentation fault. Using a signal handler to handle the SIGSEGV we were able to resolve this problem.

Results

Hardware Time (ms)	S/w Overhead Estimated (ms)	Total Estimated Time (ms)	Measured Time (ms)
10	5	15	37

Results Discussion

We expect the page fault time at 15 ms. For a page fault 1 page (4KB) has to be loaded from disk. This time should be considerably larger than the time taken to load 4KB from memory. Our measured time was consistent with this hypothesis.

Comparison with the latency of accessing a byte in main memory

Reading 4 KB from main memory takes 0.056 ms whereas reading 4 KB through page fault was found to take 37 ms. Thus we can say that reading a byte from main memory is much faster than accessing a byte through page fault.

References for RDTSC used in experiment code:

1. *http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf*
2. *https://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf*
3. *http://www.unix.com/programming/81639-rdtsc-use-c.html*