## 1. Introduction

In working with systems, Operating Systems connect software/user level with the hardware/machine level, in order to better utilize the hardware resources to perform different tasks. There are several Operating Systems with different features and characteristics to reach the performance goal of utilizing resources. In our project, we want to use the different tools to test and determine the performance of our choosing Operating System. After getting the results, we will try to use our abstract knowledge to analyze their performance and provide some conclusion behind the data. In particular, we want to test the following Operating System aspects: CPU Scheduling, Memory, Network, File System. For each aspect, we will develop testing for several smaller features, which will be described in detail later.

In our project, we plan to use the tools provided by Intel Benchmark to test the main features of our Operating System. And we plan to use mainly C/C++ to develop API for testing.

## 2. Machine Description

Processor Model: $3.1GHz,\ Dual - core,\ Intel\ Core\ i5$
CPU cycle times: $\frac{1}{3.1G} = 0.322\ ns$
CPU cache size: $L2\ has\ 256KB\ for\ each\ core,\ L3\ has\ 4MB$
DRAM type: $LPDDR3$
DRAM clock: $2133\ MHz$
DRAM capacity: $16GB$
Memory Bus bandwidth: 64 bit
I/O bus type: Thunderbolt 3 / USB 3.0 (Apple T1 conroller)
I/O bandwidth: Up to 40 Gbps for Thunderbolt 3, 10 Gbps for USB 3.0
Disk type: $SSD$ (PCI-Express)
Disk Capacity: $512GB$
Disk Transfer read: $639\ MB/s\ for\ 4K$
Disk Transfer write: $437\ MB/s\ for\ 4K$
Network type: Wi-Fi (0x14E4, 0x171) 802.11
Network Card bandwidth: Up to 1300 Mbps
Operating System: $OS\ X12.4\ (macOS\ Monterey)$

# 3. Operations

## 3.1 Measurement overhead

### 3.1.1 Overhead of reading time

Methodology:

Intel CPUs have a time stamp counter to record each cycle on the CPU. According to Paoloni[1], beginning with the Intel Pentium processor, devices contain a timestamp register to store timestamp value which can be read by calling rdtsc() assembly functions. We recognize the existence of preemption and hard interrupts that might interfere with the CPU performance measurement. We decide to run these experiments at the user level and ignore the effects from these issues.

To measure the read instruction overhead, we call the rdtsc() function twice. Two rdtsc() function calls are consecutive so the returned timestamp values, marked as start and end, indicate the start time and end time of a reading process. To determine the cycle time, we conduct the experiment to sum up 10000 times of two consecutive rdtsc() and take the average to obtain a more accurate result.

Results:

Reading time overhead (cycles)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 122.2485 | 116.5212 | 106.0219 | 105.5678 | 105.9807 | 105.2017 | 105.3872 | 106.8727 | 105.3702 | 104.9513 |

Reading time overhead (microseconds)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0.0394 | 0.0376 | 0.0342 | 0.0341 | 0.0342 | 0.0339 | 0.034 | 0.0345 | 0.034 | 0.0339 |

Prediction: Todo: base hardware performance, estimate of software overhead, prediction of operation time

Discussion:

From the measurement result, reading time overhead is mostly in the range of (104, 107) cycles. We can see the first two values are out of this range, and the following results are well bounded in the range. We first guessed these two values are caused by the cache initialization process. After repeating the test procedure ten times, the results indicate that the first one or two values are always larger than the remaining values, and the following values tend to be in the same range (104, 110). Thus we conclude that the read overhead is approximately 107 cycles by taking an average of the maximum and minimum, that is 0.0345 microseconds in this system.

Reference:

[1]https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf

[2]https://en.wikichip.org/wiki/intel/microarchitectures/saltwell

## 3.1.2 Overhead of using a loop to measure iterations of an operation

Methodology:

In this part, we will measure the iterations of an operation using a loop. The test starts with storing the timestamp counter at the beginning of the loop. Then execute the loop for a fixed amount of times, iteration_time, with no instructions inside the loop, and record the timestamp counter after the iterations. Now we have received the number of cycles, total_cycle, for the whole loop, and total_cycle / iteration_time gives the average loop overhead count. To examine any abnormal result and reduce such occurrences, we perform this test 10 times to gain a better understanding, using 10000 as the iteration_time.

Result:

Loop overhead (cycles)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 4.8817 | 4.8008 | 4.7953 | 4.8109 | 4.8027 | 5.2297 | 5.1451 | 4.9025 | 4.8767 | 5.176 |

Loop overhead (nanoseconds)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1.5747 | 1.5486 | 1.5469 | 1.5519 | 1.5493 | 1.687 | 1.6597 | 1.5814 | 1.5731 | 1.6697 |

To obtain an estimation of loop overhead, we want to analyze the assembly instructions within a loop. We think for each iteration, the CPU executes the following assembly instructions:

1. Load the conditional variable v1 to the register r1.
2. Load the conditional variable v2 to the register r2.
3. Perform the condition check with v1 and v2.
4. Update v1's value.
5. Store v1's value to the conditional variable.

So we would anticipate the CPU loop overhead to be around 5 cycles.

Discussion:

According to the loop overhead result, the CPU generates around 5 cycles for each iteration of a loop. So the measured performance of loop operation is close to our predicted performance. The experiment does not include any preparation for handling any system interruptions. We use 10000 iterations for each loop to prevent the result from being largely affected by system interruptions that cannot be recorded.

## 3.2 Procedure call overhead

Methodology:

We will define 8 functions taking different arguments from 0 to 7, then calling each of them to test their overhead in terms of cycles. To access the overhead for each procedure call on function, we use rdtsc() to measure the time before and after calling the function, and calculate the difference as the overhead. To minimize the error occurring in the experiment, we will run each function 1000 times and take the average as the overhead result. And we will perform the above procedures in 5 trials to check whether there are any outliers or stays consistent.

Our prediction is that the increasing parameter will increase the overhead of the procedure calls. The reason is that in terms of the X86 machine operations, before calling the function, the system will push each parameter into the stack. Thus the increasing parameter will increase the cycles for the machine to push parameters. However, compared to the overall operations done in the procedure call, the increasing parameter may not cause the huge increase in overhead.

Result:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Trial 1 | 22.133 | 22.114 | 22.131 | 21.31 | 21.232 | 21.717 | 23.029 | 22.28 |
| Trial 2 | 24.282 | 26.137 | 23.329 | 25.199 | 23.845 | 23.244 | 25.225 | 24.269 |
| Trial 3 | 29.883 | 28.617 | 31.042 | 31.051 | 30.334 | 28.669 | 29.897 | 28.712 |
| Trial 4 | 24.426 | 24.262 | 24.247 | 23.273 | 24.362 | 23.179 | 24.275 | 24.563 |
| Trial 5 | 24.277 | 23.387 | 23.302 | 25.208 | 23.763 | 23.313 | 24.161 | 25.275 |

Discussion: Based on the above generated result about the overhead cycles for procedure calls with different arguments, it seems like the number of arguments doesn't really affect the overhead time. Or the argument increasing is too small to be considered compared to other instructions in procedure call. Further analysis will be done later for better comparsion.

## 3.3 System call overhead

Methodology

We would like to consider the following system calls: open, read, write, close, wait, exec, fork, exit, and kill, which are popular system calls in macOS. We choose to find the cost of a minimal system call open() by recording the start and end time from two rdtsc() functions calls, and perform an open() function call from the “fcntl.h” file between two rdtsc(). Similarly, one test includes 10000 times of repeating the previous procedure. And we repeat this test 10 times and select the minimum system call cost.

Result

open() system call cost (cycles)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 12268.28 | 11778.17 | 11941.38 | 13308.33 | 11288.47 | 11205.13 | 11188.44 | 11217.06 | 11430.43 | 11387.74 |

open() system call cost (microseconds)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3.9575 | 3.7994 | 3.8521 | 4.293 | 3.6414 | 3.6146 | 3.6092 | 3.6184 | 3.6872 | 3.6735 |

The cost of a minimal system call using open() is 3.6146 microseconds according to our calculation.

getppid()

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| cycles | 2054.84 | 2121.66 | 1968.35 | 1958.72 | 1959.35 | 1990.63 | 1954.99 | 1972.45 | 2179.24 | 1967.88 |
| microseconds | 0.6629 | 0.6844 | 0.635 | 0.6318 | 0.632 | 0.6421 | 0.6306 | 0.6363 | 0.703 | 0.6348 |

The cost of a minimal system call using getppid() is 0.6306 microseconds.

getpid()

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| cycles | 177.46 | 117.04 | 110.58 | 110.5 | 108.84 | 112.59 | 109.14 | 112.24 | 109.12 | 115.01 |
| microseconds | 0.0572 | 0.0378 | 0.0357 | 0.0356 | 0.0351 | 0.0363 | 0.0352 | 0.0362 | 0.0352 | 0.0371 |

The cost of a minimal system call using getppid() is 0.0351 microseconds.

Discussion

Generally speaking, the cost of procedure calls is relatively cheaper than system calls. However, some idempotent system calls like getpid() are cached by the operating system, which causes a much smaller cost. And our test result clearly shows that getpid() cost is much smaller than getppid() and open() system calls. So the cost of a minimal system call is 0.0351 microseconds.

Procedure call overhead test is still in the process, and we will present a comparison between the cost of this minimal system call and procedure call later.

We have chosen open(), getppid(), and getpid() as the system calls to be tested for a minimal cost. However, the cost of a minimal system call could be computed from any of the system calls. More tests will be performed and updated later.

## 3.4 Task creation time

Methodology

The “hbench” paper introduces its benchmark for process creation. It concludes from its experiment result that the process creation time is highly dependent on memory. And the result between the static linked executable has a different performance as dynamically linked executables. Static files are likely to be cached in the main memory before executing while the dynamic file would be loaded every time it executes. As said in the “hbench” paper, “the CPU-dependent component has grown due to the need to build and initialize jump tables for the libraries.” So when doing our test, we also take care of eliminating the CPU-dependent effect.

Considering this, we decided to first test two things for the process creation:

1. fork() a child process from the parent process
2. exec() a new process from the parent process

Before and after every function call, we will call rdtscp() to track the time elapsed. Both fork() and exec() are primitives provided by Unix to create a user process. By calling fork() in the parent process, we can create an exact copy of the parent and control the execution of that child process. Exec() would end the original process and execute a new process. Both processes involve no dynamically linked library. In the first trial, we only test the situation when all the executables are already in the main memory, and no disk read is involved.

For kernel thread creation, we utilize a similar methodology. By pthread_create system call, a new kernel thread is created, and we utilize rdtscp() to calculate the time passed.

Result:
For process creation, the result is 1138748.98 clock cycles.
For kernel thread creation, the result is 48396.168 clock cycles.

Discussion;
We predicted that the process creation overhead would be about 800000 and kernel thread creation overhead would be about 30000. The result is much larger than the expectation, and our assumption is that it might come from other system overheads that we didn’t factor out. A main problem might be the involuntary yield of CPU by a process that would cause extra context switch overhead and the inaccurate calculation of the process creation time.

## 3.5 Context switch time

Methodology:

As defined by the “lmbench” paper, context switch time consists of two parts: 1. the time to save the state of one process, and 2. the time to restore the state of another process. The Imbench paper mentioned, "the results of some benchmarks, most notably the context switch benchmark, had a tendency to vary quite a bit, up to 30%.” So the key point to getting a relatively accurate context switch measurement is to control the variability related to it. In our first trial, we applied the methodology mentioned in the “lmbench” paper. When evaluating the context switch time, we run it in a loop and take the minimum result as the benchmark.

We also apply the methodology provided by the “lmbench” paper to measure the context switch overhead. We implemented a ring of 10 processes connected with Unix pipes. By sending tokens through the pipes, the context switch is forced. Here the token passing will generate extra costs, which have to be factored out in the measuring program.

Result:
For the process context switch overhead, we get an average of 469863.476000 clock cycles.
For the kernel thread switch overhead, we get an average of 3163.360582 clock cycles

Discussion:
We predicted that the process context switch overhead would be 30000 and kernel thread overhead would be about 2000. The result is much larger than the expectation, and our assumption is that it might come from other system overheads that we didn’t factor out.

[1] http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/

## 4.1 RAM access time

Methodology:

To evaluate the RAM access time, we applied the method in the lmbench paper, which is back-to-back-load latency. As defined in the lmbench paper, back-to-back-load latency calculates each load's time, assuming that the instructions before and after are also cache-missing loads.

Intel i5 Core has three-level CPU caches (L1, L2, and L3). L1 cache is 64kb in total for each core, 32kb for the instruction cache, and 32kb for the data cache. L2 cache is 256kb, and L3 is 5MB. Thus to evaluate the latency for main memory as well as different levels of CPU caches, we will test the machine with references to data arrays with different sizes. One thing that has to be factored out is the prefetching effect from the large cache lines. We also applied the method from lmbench, introducing the usage of stride.

Thus, our benchmark calculation would be generalized as follows:

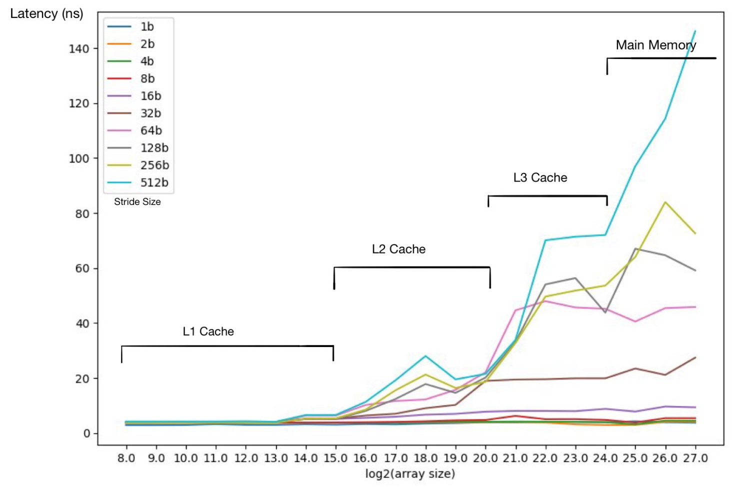
1. fix a stride size d, fix an array size n, and malloc an empty array with size n.
2. Loop through the array, the entry value at index i would be assigned at the index value of the entry that is d away from the current index i, i.e.: array[i] = (i + d)%n
3. Walk through the array for the second time, following pointer pt, where pt = array[pt], and set a timer before and after the loop, in which the program will do 100000 array references. We will get the average time for each reference.

Result

In the experiment, we tested the machine with the following parameters:

stride size(bytes): $4 \times 2^i, i \in [0, 9]$ ; array size(bytes): $4 \times 2^i, i \in [8, 17]$

The graph of latency with respect to the array size shown as follows:



From the graph, we deduce 128 bytes as a possible cache line size since it is the smallest stride size that has a great increase of latency when the array size exceeds L3 cache. so we will utilize the result when stride size = 128 bytes as our data for analysis. The following graph is our experiment result for memory access latency.

| Type | Accessing Time (nanoseconds) |
|---|---|
| L1 | 3.919 |
| L2 | 12.421 |
| L3 | 43.804 |
| Main Memory | 67.04 |

Discussion:
Before doing the experiment we have made the following estimation for the latency.

| Type | Accessing Time (nanoseconds) |
|---|---|
| L1 | 2 |
| L2 | 8 |
| L3 | 16 |
| Main Memory | 32 |

Our results for latency are comparatively longer than the normal CPU, and we estimate the problem might be that practically, there are too many background processes running that affect the memory reading time. Also, there might be some other reasons that influence the accuracy of the result, such as instruction loading time, which hasn't been factored out in our benchmark. Last, since the stride convergence is not very apparent when the array size becomes larger, we consider further diversifying the array size and stride size to get a better convergent result.

## 4.2 RAM bandwidth

Methodology:

To measure the bandwidth for transferring memory, we plan to first construct the transfer data (read/write) in bytes, and then divide it by the transfer time (measured in experiment). And we split it into two tasks: reading transfer and writing transfer. According to the Imbench paper's approach, "Memory reading is

measured by an unrolled loop that sums up a series of integers" (5) and “Memory writing is measured by an unrolled loop that stores a value into an integer (typically a 4 byte integer) and then increments the pointe” (5), we plan to allocate an array of integers with 64MB as memory reference.
For memory writing experiments, as the Imbench paper states, we simply use an unrolled loop (with 32 offsets) to store an integer value (like 1) into each index of an array. After we perform memory writing approaches on 64MB arrays, time consumption is recorded in terms of cycles using rdtsc(). The bandwidth for writing memory is simply calculated by the size of the integer array (64MB) divided by the recorded time (translated from cycles to second).
For memory reading experiments, we will reuse the allocated array which is just filled by the memory writing, and perform memory reading for every integer index of the array. Using the Imbench approach, we will use an unrolled loop (with 32 offsets) to sum up every integer in a 64MB array, and calculate the bandwidth as 64MB memory divided by measured consumption time (translate from cycles to second).

Result:

| | Memory reading bandwidth | Memory writing bandwidth |
|---|---|---|
| Trial 1 | 8.941 GB/s | 10.553 GB/s |
| Trial 2 | 9.056 GB/s | 10.561 GB/s |
| Trial 3 | 8.815 GB/s | 10.578 GB/s |

Discussion:
In our prediction, the memory bandwidth is calculated by Clock Frequency * Memory bus width (in bytes) * data transfer rate. According to the machine description, our memory’s frequency is 2133mhz, with 2x data transfer rate and $64\ bits\ /\ 8\ =\ 8\ bytes$ Memory bus bytes. As the Imbench paper said: “the processor cost of each memory operation is approximately the same as the cost in the read case”, so we expect both reading and writing to have the same memory bandwidth. Thus in our prediction, the memory bandwidth for both reading and writing should be $2133mhz\ \times 2\ \times 8\ =\ 34128\ Mb/s$ $=\ 34.13\ GB/s$. Since there will be efficiency loss for software overhead in practice compared to theoretical results, we predict there will be like 50% efficiency loss, resulting in like 17 GB/s for bandwidth.
However, after testing for both reading and writing memory bandwidth, the results are much lower than our prediction. I think there are many factors that prevent our experiment from getting the maximum bandwidth, like many other kernel applications will occupy resources when executing the experiment, and our machine is too old (5 years from purchase) which will cause hardware loss. In addition, my unrolled

loops method is only with 32 offsets, which may be optimized by using some algorithms. I will continue to optimize the unrolled loop algorithm to get more accurate results from memory bandwidth.

## 4.3 Page fault service time

Methodology

A page fault occurs when a process currently executing tries to access a page which is not in the memory. Considering the three types of page fault, minor/major/invalid page fault, we want to focus on the major page fault in this experiment. A major page fault is a mechanism used by the OS to increase the amount of available memory on demand. It occurs when the page being accessed is not in the main memory but is still a valid address.

To measure the time for faulting a page from disk, we create a large file with size of 10MB and use mmap() to generate the mapping between the memory and and the disk. We use a stride of 40KB (10 pages), and access the memory mapped file 100 times. With this being a test, we repeat this test 10 times to receive an average result number. Notice that before each test, the disk buffers should be clear in order to get page fault, the purge instruction is executed before each test.

Estimation

The whole process is mainly disk reading time and the OS overhead. The disk reading time can be estimated by the page size divided by the write rate, which is approximately 0.009 ms. OS overhead might include the context switch and syscall. From the previous experiment results, the approximation is 0.15 ms.

Result

Measured average page fault service time = 94157.5 * 0.322 ns = 0.303 ms

Discussion

The test result is 0.303 ms, which is less than the estimation result. This comparison result might be caused by other overheads out of our current consideration.