CSE 221 SYSTEM MEASUREMENT PROJECT

Author: Yu Xia, ChenFu Xie

December 15, 2014

Contents

1. *Introduction*

Source code link: https://github.com/dotcom900825/cse221

Operating System plays a vital role in between the software and hardware system, it controls and manage hardware resources and provide the interface for software system to be able to fully utilize them. In order to better build and understand the Operating System, we need to have a good understanding of the influences and constrains brought by the underlying hardware components(CPU, RAM, Storage, Network, etc.)

Through this project, we want to dig through the underlying hardware's detailed characteristics and create a program to perform a series of experiments to help us analysis, test and justify the Operating System's performance.

The main goal of this project is to apply a different sets of tests to benchmark the Operating System, then perform analysis and toward the results. We plan to utilize C/C++ to use different system call API and Python for network related experiment to build the benchmark software.

1.1 Machine Description

Here is the detailed description of the test machine.

Hardware	Spec
CPU Model	Intel Core I5, 2.6GHz, dual cores
Cycle Time	(1 / 2.6G) = 0.38 ns
RAM size	8GB
L1 Cache	Each core has a 32KB Instruction Cache and a 32KB Data Cache
L2 Cache	Each core has its own dedicated 128k level 2 cache
L3 Cache	System has 768KB of shared level 3 cache
Instruction Set	64bit
Memory Bus	Type: DDR3 Speed: 1600MHz Width: 64bit
I/O Bus	SATA (AHCI Version 1.30 Supported)
Disk	Type: SSD Capacity: 256GB Media Name: Apple SSD SM0256F Uncached Write: 131.07 MB/sec [4K blocks] Uncached Write: 387.52 MB/sec [256K blocks] Uncached Read: 20.30 MB/sec [4K blocks] Uncached Read: 273.92 MB/sec [256K blocks]
Network Card	Wifi Interface: Airport Extreme(0x14E4, 0x112)
Operating System	OS X 10.10

IO results generated by: Xbench (http://xbench.com/)

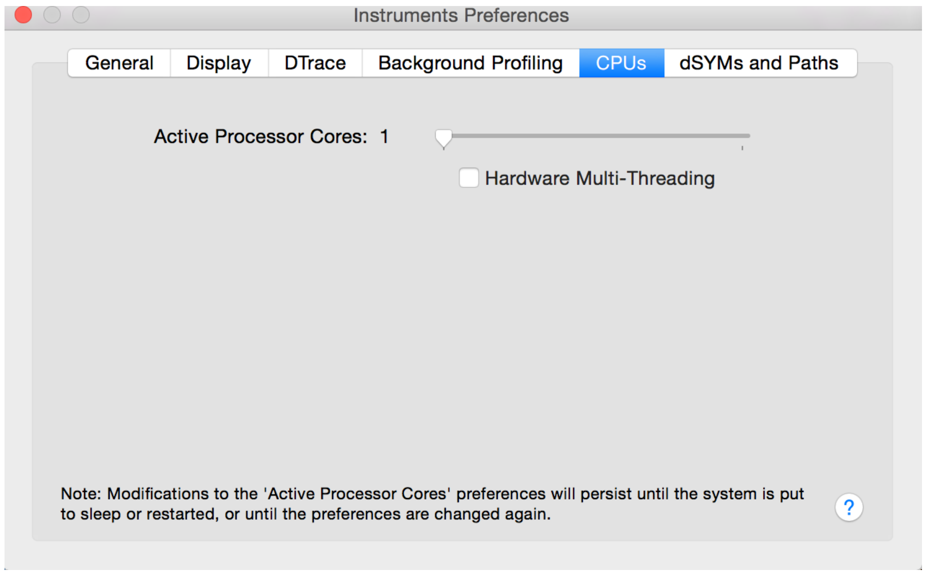
1.2 Testing Environment Setup

In order to accurately profile the system and its underlying hardware through our benchmarking tool, we need to properly setup the testing environment before we run the program and collect the data.

Given the fact that we are running on a multi-core system, and running a number of other processes at the same time under some specific OS scheduling policy, how to rule out those interferences from multi-core, hardware multi-threading, undesired process context switches becomes our preparation work before we dive into the actuall experiments.

1.2.1 Single Core Environment Setup

Limiting CPU Cores on-the-fly in OS X is quite straightforward, the XCode toolkit provides a diagnostic tool for debugging and profiling OS X apps, and one of the tool they provide is to run the test program on a limited system. We also switched off the hardware multi-threading feature.



Active Processor Cores: 1

Hardware Multi-Threading

Note: Modifications to the 'Active Processor Cores' preferences will persist until the system is put to sleep or restarted, or until the preferences are changed again.

Reference: http://jesperrasmussen.com/blog/2013/03/07/limiting-cpu-cores-on-the-fly-in-os-x-mountain-lion/

1.2.2 Preventing Multi-Process Context Switches

Modern OS would normally run multiple processes in the same time, and apply certain type of scheduling algorithm to manage them. In order to minimize the impact of processes context switch on the benchmark program, we need to execute the program with an altered scheduling priority. In Unix world, there a command line program called "nice" that can allow super user to run utilities program with priorities higher than normal. In our experiment, we will kill all the other irrelevant and background processes and run the benchmark program with the parameter of -20 which gives it the highest priority.

References:

- http://en.wikipedia.org/wiki/Nice_(Unix)
- http://linux.die.net/man/1/nice

1.2.3 Timestamp Reading Mechanism

We realize that a lots of the benchmark functionalities would require accurate timing calculation to allow us perform good measurement. So it is vital for us to have some utility function that we can use for timestamp reading. For our project, we utilize the Time Stamp Counter(TSC) which is a 64-bit register present on all x86 processors including our testing machine. It will give you the number of clock cycles since the CPU powered up or reset. Since RDTSC can be executed out-of-order, so we should flush the instruction pipeline to prevent the counter from stopping measurement before the code has actually finished executing.

```
static inline uint64_t rdtsc(void) {
  uint32_t lo, hi;
  __asm__ __volatile__("xor %%eax, %%eax;" "cpuid;" "rdtsc;": "=a" (lo), "=d" (hi));
  return (((uint64_t)hi << 32) | lo);
}
```

References:

- http://en.wikipedia.org/wiki/Time_Stamp_Counter
- http://www.cs.wm.edu/ kearns/001lab.d/rdtsc.html
- http://www.strchr.com/performance_measurements_with_rdtsc

2. *CPU Operations*

2.1 Measurement Overhead

2.1.1 Time Stamp Counter Read Overhead

Methodology

In this section, we will measure the read overhead for the timestamp register by calculating the average cpu clock cycles spend between two RDTSC calls. In between the two consecutive RDTSC calls, we make sure there are no other instructions get executed. We sampled the read instruction for 10000 times, and sum up the total elapsed cycles, then calculate the final average result.

Prediction

Through some technical reports and online articles discussing how to use RDTSC instruction for performence monitoring. We find out that the overhead of RDTSC is about 100 - 150 cycles.

Reference:

- https://www.ccsl.carleton.ca/ jamuir/rdtscpm1.pdf
- http://www.strchr.com/performance_measurements_with_rdtsc

Test Results

	1	2	3	4	5	6	7	8	9	10
Cycles	110.01	186.56	104.9	107.7	113.10	206.44	112.30	104.97	103.79	102.58

Median	Standard Deviation
107	3.99

Analysis

From the results above, we can see that the second counter count are relatively high, that's probably due to cache warm up, after that, the count pretty much stablize around 100, than it bumps up to 206 suddenly, probably due to context switch, then drops down to around 100 again. So the benchmark result is pretty match up with intel's spec.

2.1.2 Loop Overhead

Methodology

In this section, we will measure the program loop overhead. In order to measure the cpu overhead cycles for each loop, we save the cpu counter first then run the loop with no instructions within for 10000 times. After this, we read the cpu counter again and calculate the difference. So we have an accumulative cycle count for that 10000 times, and we divide it for 10000 to retrive the average loop overhead cycle. We performed the test for 10 times to compare the results.

Prediction

For each loop, we think there are about 6 steps would happen in the CPU.

- Load the conditional operand to register
- Check the value against looping condition
- Change the conditional operand's value
- Store the operand's value to memory
- Calculate inner loop program's address
- Load address to Program Counter

So we predict it will be about 6 cpu cycles for each loop

Result

	1	2	3	4	5	6	7	8	9	10
Cycles	6.52	5.32	7.18	5.32	7.16	6.39	5.32	6.00	5.32	8.20

Median	Standard Deviation
6.00	1.006

Analysis

As the results indicate, it is very close to our prediction. Besides, the deviation is also small enough to convince us that the loop overhead is around 6 CPU cycles.

2.2 Procedure Call Overhead

2.2.1 Methodology

In order to calculate the procedure call overhead, we want to measure how the overhead increases as the function call's parameters grow. Thus we created 8 test functions with different number of parameters, ranging from 0 to 7. We record the time before and after calling the function for 10000 times, then calculate the average.

2.2.2 Prediction

In a C function call, the following steps will happen in sequence.

1. Push parameters to stack in reverse order.
2. Push return address to the stack.
3. Jump to the start of the function.
4. Push local variables to the stack.
5. Place the return value, if any.
6. Pop the local variables off the stack.
7. Jump to the address at top of stack.
8. Pop return address and parameters off stack.

In step 1, different number of parameters will make the function call to take different number of cycles. If the function have n local parameters, while no local variables exist in our examples, the whole process would take about n + (8-10) cycles.

2.2.3 Result

P size	t1	t2	t3	t4	t5	t6	t7	t8	t9
0	122.42	107.59	113.43	124.28	109.77	112.48	132.99	112.11	130.26
1	124.36	118.44	122.42	122.168	115.81	119.40	135.04	114.09	118.55
2	113.20	120.01	121.95	119.02	118.80	120.19	125.25	115.45	117.46
3	119.03	118.80	118.86	120.64	118.60	115.91	116.15	116.27	115.54
4	134.28	114.56	116.72	118.01	123.38	115.91	121.56	126.58	120.15
5	124.95	121.07	124.91	122.50	113.82	113.70	119.50	122.34	114.21
6	143.82	118.59	120.41	117.99	128.96	193.45	124.27	117.71	130.88
7	124.01	124.35	122.35	122.63	121.85	120.88	127.48	127.69	117.15

P size	Median	Standard Deviation
0	118.37	9.31
1	121.14	6.13
2	118.80	3.19
3	117.28	2.28
4	121.24	6.2
5	119.43	4.43
6	132.90	24.21
7	132.90	24.21
8	123.54	3.30

2.2.4 Analysis

After subtract the read overhead from the average results, we can find the actual outcome is also pretty close to what we have predicted. And the cycle counts do increase with the increase of the number of parameters.

References: http://classes.soe.ucsc.edu/cmps012b/Spring97/Lecture06/sld005.htm

2.3 Task Creation Time

2.3.1 Process Creation Overhead

Methodology

In order to create new process, we use system call fork() to spawn the child process. Because we want to only measure the creation overhead, so in our program we will make the child process exit immediately. The test will be repeated for 10000 times and calculate the average cpu cycles.

Prediction

When we try to create a new process, there is a lot of steps for the OS to take including assign an unique process id to the child process, allocate resources and space to the child process, add new entry to the process control block, and set up approriate linkages. In sum, we believe the process creation is a quite time-consuming job, we predict it will cost about 500000 to 600000 cycles.

References: http://cse.unl.edu/ witty/class/csce351/material/lecture/ppt/lecture3.ppt

Result

	1	2	3	4	5
Process Creation	897842.35	878906.99	870298.15	869315.94	882059.20

Average	Standard Deviate
878484.43	9624.98

Analysis

From the test result we can see that process creation time is quite big, and higher than our original estimation. This might due to the measured cycles also include the context switch time.

2.3.2 Thread Creation Overhead

Methodology

We use the POSIX thread libraries to create new kernel level threads. The test process is similar to what we have used to test the process creation overhead. We adopted pthread_create() to spawn new kernel thread, and making the newly created thread exit immediately, thus we can measure the exact thread creation overhead. We repeated the test for 10000 times and calculate the average.

Prediction

Compared to the process creation, thread creation is a relatively more light weight task. Because all the created threads share the same memory resource. Thus we guess the time takes to create a new thread will be about 50000 to 60000 cycles.

Result

	1	2	3	4	5	6	7	8
Thread Creation	50831.90	54383.20	48697.33	58634.83	56372.63	58938.37	52862.80	52833.51

Average	Standard Deviate
54867.22	3948

Analysis

The thread creation overhead pretty much match up with our prediction, while the process creation overhead is higher than what we have expected. We think it might due to the context switch overhead between different processes.From the results we can also find the process creation overhead is almost 10 times of the thread creation overhead which we believe is also reasonable and can be considerate as accurate result.

Reference:

- http://en.wikipedia.org/wiki/Thread_(computing)
- http://stackoverflow.com/questions/2267545/why-are-threads-called-lightweight-processes
- "Operating Systems Concepts" by Silbershatz, Galvin and Gagne

2.4 Context Switch Time

2.4.1 Process Context Switch Overhead

Methodology

In order to measure the process context switch overhead, we need to first use fork() to create a child process. Once the child process is created then in the parent process we use RDTSC call to record the starting time, then call wait() to invoke the context switch. In the child process, we first call RDTSC to capture the context switch end time, then we return the clock read back to the parent process through UNIX pipe. And we calculate the elapsed cycle, and perform the test multiple times to calculate the average.

Because we set the test to run on a single processor and make our test program to run in the highest priority, the child process will high likely be chosen as the next process to run.

Prediction

During the process context switch, there will be a switch from user mode to kernel mode and change the address space from the parent's one to the child's one which is a relatively heavy workflow. So we predict this would take about 300000 - 400000 cycles which is around 110 - 150us to complete the whole process.

Result

	1	2	3	4	5	6	7	
Process Context Switch	171.48us	158.27us	172.00us	162.13us	168.90us	166.06us	180.86us	163

Average	Standard Deviate
167.2us	6.545

Analysis

The test results are pretty matched up with our prediction and confirmed that context switch is a relatively high overhead operation.

Reference:

- http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html

2.4.2 Thread Context Switch Overhead

Methodology

Similar to what we did in the process context switch test, the difference here is that we use pthread_create() to spawn a new thread, and use pthread_join() to force the thread context switch.

Prediction

Compared to the process context switch, the thread context switch is faster. Since all threads share the same resources and there is no execution mode change, so we estimate the cpu cycles take to complete the thread context switch is about 5000 - 6000 cycles.

Result

	1	2	3	4	5	6	7	8
Thread Context Switch	5521.03	5697.63	5924.99	5582.45	5465.382	5454.258	6257.314	5412.49

Average	Standard Deviate
5924	826

2.4.3 Analysis

The measured result pretty much matched up with our estimation, compared to the process's context switch time, thread context switch time is much more faster , about 50-60 times faster than the process's one. Compared to process's context switch, for the thread context switch the kernel doesn't need to change the current virtual memory space, and won't flush out the TLB during thread context switch

Reference:

- http://stackoverflow.com/questions/304752/how-to-estimate-the-thread-context-switching-overhead

3. *Memory Operations*

3.1 RAM Access Time

3.1.1 Methodology

In this section, we will measure the RAM access time, particularly the latency for individual integer accesses to main memory and the L1 and L2 caches. According to the lmbench paper, there are four most common definitions, memory chip cycle time, processor-pins-to-memory-and-back time, load-in-a-vacuum time, and back-to-back-load time. In our experiment, we will focus on measuring the back-to-back-load time, which is the time that each loads takes, assuming that the instructions before and after are also cache-missing loads.

To measure the integer accesses latency, we will instantiate different chunks of memory spaces and fill them up with integer array. Each array is different in terms of size ranging from 4KB to 512MB. Since L1, L2 and L3 cache on the test machine are 32KB, 128KB and 768KB respectively(Single Core's setting), 4KB to 512MB should be enough to portrait different memory hierarchies's access latency.

Following the same strategy as the lmbench paper adopted when experimenting the memory access latency, we use different stride size for integer access on different size of arrays in order to experiment the possible effect caused by cache line sizes and memory prefetching.

Here are the steps we took to measure the memory access latency for different size of arrays and stride sizes.

1. According to the lmbench paper, there are two parameters existing in the experiment, array size and array stride. For each array size, we need to create a list of pointers for different strides. Here are the pseudocode snippets

```
for(int i = 0; i < array_size; i++){
  array[i] = (i + stride_size) % array_size
}
```

2. Use RTDSC call to record memory access cycles, in order to measure the back-to-back loads, we need to make sure the instructions before and after are

also cache-missing loads. To achieve that, we use the following code snippets. This code also avoid prefetching. Since the next array access index is stored in the current accessed element's value.

```
next_index = 0
total_time = 0
start = rtdsc()
for(int i = 0; i < STEP_NUMBERS; i++){
  next_index = array[next_index]
}
end = rtdsc()
total_time = end - start

access_time = (total_time - memory_read_overhead) / STEP_NUMBERS - loop_overhead
```

3. We repeat the experiments for 1000 times and calculate the mean of the cycles for different array sizes under certain stride steps.

Prediction

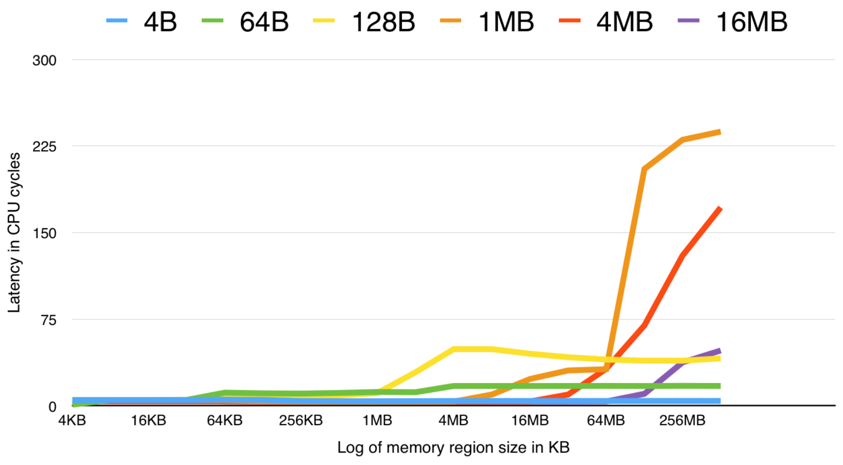
We predict the memory access latencies as following

- L1 cache: 4 cycles = 1.53ns
- L2 cache: 10 cycles = 3.84ns
- L3 cache: 40 to 80 cycles = 15.38ns to 30.76ns
- Local DRAM: 120 cycles = 46.15ns

Test Result

Here is the test result we obtained by running the experiment with stride size 128 Byte. We subtract the RTDSC read overhead and loop overhead from the raw data.

Memory Region	Mean Memory Access Latency(cycles)	Corresponding time(ns)
4KB	5	1.92
8KB	5	1.92
16KB	5	1.92
32KB	5	1.92
64KB	9	3.46
128KB	9	3.46
256KB	8	3.07
512KB	9	3.46
1MB	11	4.23
2MB	29	11.15
4MB	49	18.84
8MB	49	18.84
16MB	45	17.30
32MB	42	16.15
64MB	40	15.38
128MB	39	15.0
256MB	39	15.0
512MB	41	15.76



Analysis

From the results above, we'd like to analyse and interpret the data and try to reason about our finding.

When the stride size is 4B, which is the size of one integer. In this case, according to our program the next array index to visit is the one next to the current visited index. So all the memory access will happen in L1 cache since the cache line is 64 bytes and can fit in 16 integers consecutively, except for some rare circumstances when next index is not within the current cache line, then it needs to be fetched from low level cache or main memory. This is why the outcome is rather small and stable when stride size is 4B.

When the stride size grows to 64B and 128B, which are bigger than one cache line size but smaller than the entire L1 cache, this will cause each array element access in different cache lines when the test array's size is big enough to hold a large amounts of array stride, however, when the array size is smaller then the stride size, our implementation would actually fall back to consecutive array access. So when the test array size expand from 4KB to 32KB, which is smaller or equal to L1 cache size and smaller then the stride size, the access time will be small and stable, similar to 4B stride time.

When the array size grows from 64KB to 128KB, which is smaller or equal to L2 cache size, the access time will be stable but bigger than the previous ones, since we have more L2 access. When the array size is from 256KB to 512KB, which are smaller than L3 cache size, the array has to visit more L3 cache than before, this is why the access time becomes a little bigger again. Finally, the array size out growth the L3 cache, thus we have to fetch from main memory, which results a even longer access time.

When the stride size is from 1MB to 16MB, which is bigger than the 768KB L3 cache size and cache line size. With the array size grows from 4KB to 2MB, but still lesser than the stride size, will cause the next index to visit to roll back, which is actually next to the index we just visited, because we calculate the next index to visit as (i + strideSize) MOD arraySize = i. This is why all these access times become quite stable. After the array size grows bigger than the stride size, such as 16MB, 32MB, etc, which is also bigger than the L3 cache size. Each access to next index in the array will cause cache miss and render a longer memory access time

Reference:

- http://igoro.com/archive/gallery-of-processor-cache-effects/
- https://www.usenix.org/legacy/publications/library/proceedings/sd96/full_papers/mcvoy.pdf

3.2 RAM bandwidth

3.2.1 Methodology

Memory bandwidth represents the rate at which the data can be either read or stored to the memory by the processor. In order to profile the bandwidth, we followed the lmbench paper's approach and measured the ability to copy, read, and write data over a varying set of sizes. Since we need to touch the main memory to get a accurate measurement, the step size has to be bigger then the cache line size to be able to avoid cache hit. Once we have the time measurement on accessing the array, we can calculate the bandwidth by the following formula:

$$Bandwidth = \frac{Numberofbytesread/written}{DataTransferTime}$$

To measure the read and write bandwidth separately, we allocated a 1GB of memory region for integer array and perform read and write operation repeatedly. The first attempt we had is to repeatedly access the array and write to every element in the array. This method, however, may suffer from touching the system cache and read memory into the cache which will increase the memory traffic and obfuscating the memory write test result. In order to tackle this issue, we adopted another strategy Non-temporal instructions from Ulrich Drepper's paper "What every programmer should know about memory", these instructions do not touch the cache line and modify it, instead the new content is directly write to the memory. In addition to that, we also came across another great blog explaining how he performed the memory bandwidth measurement, and eventually he used the repeated string instructions to further improve the results. We performed all those three tests and recorded the results.

3.2.2 Prediction

According to the DRAM spec and memory bandwidth definition

Bandwidth = DRAM Clock frequency * memory bus width * number of lines

For our testing machine's DDR3 memory, we have

- Clock frequency: 1600MHz
- Number of lines: 2
- Width: 64bits

Using these numbers we can calculate the test machine's memory bandwidth as 1600Mhz * 64 / 8 * 2 = 25.6GB/s

3.2.3 Result

Memory read result

Read Bandwidth Prediction	Loop Memory Read
25.6GB/s	14.57GB/s

Memory write result

Write Bandwidth Prediction	Loop Memory Write	Non Temporal Write	Repeated String Instruction
25.6GB/s	8.51GB/s	16.38GB/s	18.91GB/s

3.2.4 Analysis

We noticed that the normal loop read and write operation generated a relatively low bandwidth, almost always under half of the theoretical memory bandwidth. We think the main problem is due to the cache read & write policy. For modern cpu, when we perform write operations, when the data size is below 64 bytes(Cache line size), the cache must first read the entire cache line from the main memory and modify it. Then write to the main memory when the operation is over. This means although we only want to measure the write bandwidth, the operation itself actually will cause both memory read and write, which make the result bandwidth

References: http://codearcana.com/posts/2013/05/18/achieving-maximum-memory-bandwidth.html
http://www.akkadia.org/drepper/cpumemory.pdf

3.3 Page fault service time

3.3.1 Methodology

A page fault is a trap to the software raised by the hardware when a program accesses a page that is mapped in the virtual address space, but not loaded into the physical memory. There are mainly three different kinds of page faults, but in this experiment we will primarily focus on the "major page fault". This kind of page fault occurs when the page being referenced is not in the main memory but is still a valid address. In this case, the page fault handler will need to bring this page from the disk to the main memory. And if there is no free memory frames available, the operating system will have to evict one existing pages of the current process from the main memory under some specific strategies, LRU in most cases.

In the experiment, we first create a relatively large file(3.2GB in our setting) filled with random characters. This file will be memory mapped using the mmap() function. Then we access 100 times to the memory mapped file with 16MB array stride.

When we performed the experiment, we noticed that the first time run always generate a number of magnitude higher result, we believe is caused by the system cache, so every time before we perform the tests, we will execute the "purge" command to force clear out the cache memory

The time is measured using RDTSC call before and after all the accesses. Besides, we subtracted the loop overhead from the total cpu cycles, and we performed the experiment for 10 times then average out the results.

3.3.2 Prediction

The Page Fault latency in hardware side is mainly the disk access time and transfer time. According to the reference, we estimate the access time of the

SSD is about 0.2ms and the read speed is about 270MB/s(See machine description). Since the page size is 4KB, the transfer time is about 0.014ms. We also need to take the software overhead into consideration, including saving the current context, context switching, accessing the page from the file, context switching again, and loading the current context. We estimate the above processes will take about 0.1ms. So the whole Page Fault time is estimated to be about 0.314ms.

Predicted time(Hardware + Software)	Measured Mean Time	Standard Deviation
0.314ms	0.491ms	0.013ms

3.3.3 Analysis

The measured outcome is pretty close to our prediction with a very low standard deviation. Thus we can believe the results is accurate.

From the result we can say that the page fault operation cost a lots of cpu cycles. When we try to access 1B from the disk, the time = Page Fault Time / Page Size = 0.103us. However, when we try to access the same data from the main memory it takes about only 22.985ns. From this comparison, we can see even access to the SSD is much slower than access to the main memory.

References: http://en.wikipedia.org/wiki/Page_fault
http://man7.org/linux/man-pages/man2/mmap.2.html

4. *Network*

4.1 Remote Machine Description

Here is the detailed description of the remote test machine.

Hardware	Spec
CPU Model	Intel Core I5, 2.4GHz, dual cores
Cycle Time	(1 / 2.4G) = 0.41ns
RAM size	8GB
L1 Cache	Each core has a 32KB Instruction Cache and a 32KB Data Cache
L2 Cache	Each core has its own dedicated 256KB level 2 cache
L3 Cache	System has 3MB of shared level 3 cache
Instruction Set	64bit
Memory Bus	Type: DDR3 Speed: 1600MHz Width: 64bit
I/O Bus	SATA (AHCI Version 1.30 Supported)
Disk	Type: SSD Capacity: 256GB Media Name: Apple SSD SM0256F Uncached Write: 131.07 MB/sec [4K blocks] Uncached Write: 387.52 MB/sec [256K blocks] Uncached Read: 20.30 MB/sec [4K blocks] Uncached Read: 273.92 MB/sec [256K blocks]
Network Card	Wifi Interface: Airport Extreme(0x14E4, 0x112)
Operating System	OS X 10.10

4.2 Round trip time

4.2.1 Methodology

In this section, we will measure the data round trip time to profile the networking performance. For both loopback and remote results, we try to predict and evaluate the baseline network performance and the overhead of OS software.

To measure the RTT, we created a client and a server program that use TCP socket for sending/receiving data packets. The client will first try to establish TCP socket connection with the server side, once the connection is established, the client will send out a 32 bytes of data in a single packet to the server and we ensure the network packet size equals to ICMP packet size. Upon receiving the data, the server will acknowledge it by sending the same 4 bytes of data back to the client within another data packet. We repeat the data sending and receiving for 100 times and measure the elapsed time occurred on the client-side. We performed the experiments for 100 rounds and records all the results.

For the remote results, we run the server program on another experiment machine and the client on local environment, both of them running under the same wireless network and communicate through the known ip address. For the loopback results, in the client program we change remote ip address to localhost and run both the client program and server program on the same machine.

To compare our measured results, we also ran the traditional "ping" command 100 times and collect the results on remote and localhost setup.

Prediction

Round trip time indicate the length of time it takes for a signal to be sent plus the length of time it takes for an acknowledgment of that signal to be received. Most of the OS provides a "ping" command to measure the round-trip time. For the loopback time's prediction, since it would only touch the loopback device which is a virtual network interface and entirely handled through the routing table, the ping packets don't pass through any physical network interface, so there is no hardware overhead at all and it should be pretty fast, reasonably estimate would be around 0.1 ms . For the remote interface, the data packet needs to go through the network card, routers in the WIFI network, server's network card and travel back. The link speed of our testing WIFI network is about 144Mbit/s, so the ideal RTT for transferring 64 bytes data one way round would only take less than 0.1ms, but actually we estimate it normally would take around 10 - 25ms to reach another computer within the same ethernet. Most of the latency is introduced by router lookup and packet forwarding.

Test Result

Here is the test result we obtained by running our test program and running the built in "Ping" command.

	Estimated overhead	Actual Ping time	Measured time	Standard deviation
loopback interface	0.1ms	0.119ms	0.082	0.0402
remote interface	10 - 25ms	4.434ms	5.278ms	4.478

Analysis

We measured the round trip time by using the Ping command and the communication between the simple client and server program we implemented. From the result, we can see that the RTT measured by Ping command is around 32.171ms, while the RTT measured by our program is about 35.708ms which is slightly slower than the Ping command result.

We believe that the additional overhead might comes from two places. First, ICMP protocol is part of the IP protocol suite, so ICMP packets are handled at kernel level on both side and didn't rely on TCP protocol at all which makes it faster. However, our server and client program are running at user level, although through wireshark we make sure the ethernet packet size for Ping command and our program's is the same, but since our program need to utilize the Tcp and IP protocol thus adds more overhead. Another reason is because the user level process need to trap into kernel mode and make system calls to send or receive data packets.

Overall, our program achieved the similar results with low standard deviation so we are pretty satisfied.

Reference:

- http://rescomp.stanford.edu/ cheshire/rants/Latency.html
- http://en.wikipedia.org/wiki/Internet_Control_Message_Protocol
- http://en.wikipedia.org/wiki/Ping_(networking_utility)

4.3 Peak bandwidth

4.3.1 Methodology

To measure the peak bandwidth, we still use the simple server and client program written in Python and perform data transfer operation. A typical method of performing a measurement is to transfer a large file from client to the server and measure the time required to complete the transfer. Since we want to measure the peak bandwidth, so we want to send out as much data packets as we can at the same time. We also measured the peak bandwidth by sending different size of big data chunks ranging from 2MB to 128MB to measure the impact resulted by different file size.

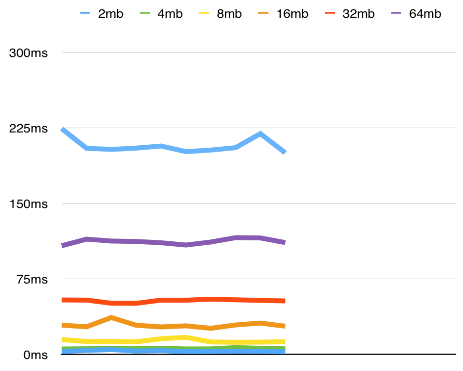
4.3.2 Prediction

The peak bandwidth can be calculated by using the max tcp window size divide the round trip time for the path. For the loopback interface, we have average RTT for 0.097ms and the TCP window size as 64KB, then we can calculate out the predicted peak bandwidth for loopback interface as 675MB/s.

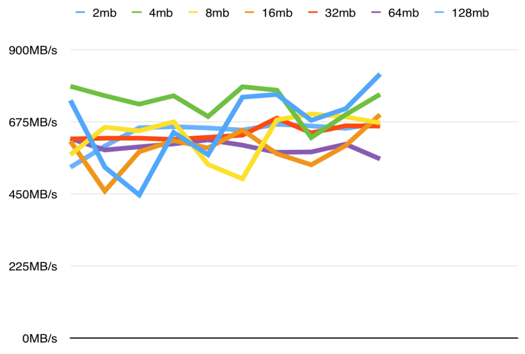
For the remote interface, we have average RTT for around 4.4ms, then we can calculate out the predicted peak bandwidth as 15.10MB/s

4.3.3 Test Result

	Estimated peak bandwidth	Measured bandwidth	Standard Derivation
loopback interface	675MB/s	643MB/s	0.45
remote interface	15.10MB/s	4.36MB/s	0.68

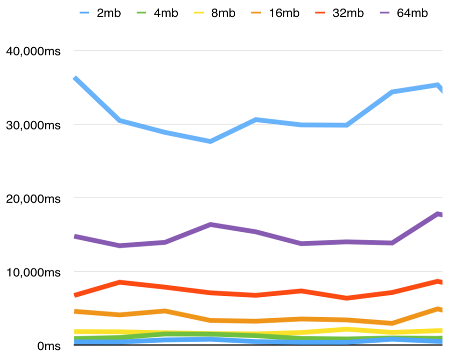


(a) loopback_time

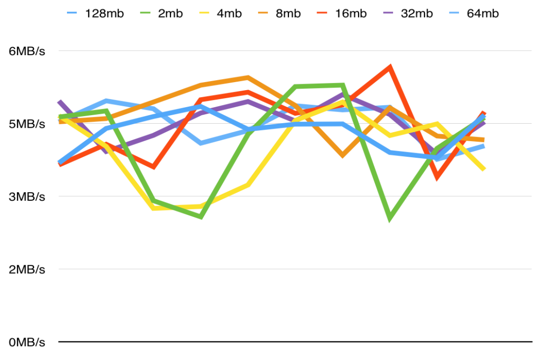


(b) loopack_bandwidth

Figure 4.1: loopback time and loopback bandwidth



(a) remote_time



(b) remote_bandwidth

Figure 4.2: remote time and remote bandwidth

4.3.4 Analysis

Through Mac OS's network utility, we know that the peak WIFI link speed for UCSD protected network is 300Mbit/s and the predicated bandwidth for 103Mbit/s while the peak bandwidth we measured is 34.88Mbit/s. There are several possible reasons that result this disparency.

First, since we conducted the experiment on the UCSD-PROTECTED wifi in Geisel library, which is shared by many other people, we anticipated a lots of network contention even collisions when we try to conduct the experiment and have impact on the peak bandwidth we tried to measure

Second, since we use the TCP packets to transfer between server and client. There is slow start in TCP transmission, which will not send that much data that could use up the whole channel. Moreover, client will cut the send window by half once it doesn't receive one packet, no matter this is caused by the overwhelming of the channel or by the unstable of the wifi connection. And the throughput is also constraint by the size of receive window in sender side. Moreover, TCP will have to retransmit packets that are lost or corrupted. Besides, there are a lot of overhead due to the headers of different protocols, 20 byte of IP header for example. All these factors stated above will reduce the bandwidth we measured.

Finally, when we measure the data size sent between the client and server, we only considered the "goodput" which is the number of useful information bits delivered excludes protocol overhead bits as well as retransmitted data packets. So the goodput is always lower then the peak throughput bandwidth. Several factors that would cause lower goodput then throughput.

- Protocol overhead
- Transport layer flow control and congestion avoidance
- Retransmission of lost or corrupt packets

References:

- http://bradhedlund.com/2008/12/19/how-to-calculate-tcp-throughput-for-long-distance-links/
- http://en.wikipedia.org/wiki/Measuring_network_throughput
- http://en.wikipedia.org/wiki/Goodput

4.4 Connection Overhead

4.4.1 Methodology

To establish a connection, TCP protocol uses a three-way handshake. First, the client send a SYN to the server. In response, the server replies with a SYN_ACK. Finally, the client sends an ACK back to the server. At this point, a full-duplex communication is established.

The TCP connection termination theoretically uses a four-way handshake, with each side of the connection terminating independently. when the client wishes to stop its half of the connection, it transmits a FIN packet, and server acknowledges with an ACK. It is also possible to terminate the connection by a 3-way handshake, when the client sends a FIN and the server replies with a FIN&ACK and client replies with an ACK.

In the experiment, on the server side we first create a socket using the socket() system call, tied it to a port and use the listen() system call to wait for any connection from the client. While the client uses connect() to establish connection to the server at the specified IP address. We measure the timestamp before and after this call and repeat for 100 times, calculating the average TCP setup time. For the connection tear down overhead measurement, it is similar to this setup as well.

4.4.2 Prediction

Connection Setup

- Remote interface: For the TCP connection setup, we need to perform three way handshake which would result 1.5 times longer time then RTT we measured, so the prediction will be around 6.5ms - 8ms
- Loopback interface: For the loopback interface, since the packets does not go through the actual network, so based upon the loopback RTT we measured before, we estimate that the loopback interface connection setup time would be around 0.18ms

Connection Teardown

- Remote interface: For the TCP connection teardown, the client side needs to send out FIN packet and doesn't need to wait for the server's acknowledgement. Most of the overhead would due to TCP stack's processing overhead, which would be within 1ms.
- Loopback interface: For TCP connection teardown overhead, similar to the remote interface, the magnitude should be way smaller then the RTT, so our prediction still would be around 1ms.

	Prediction	Measured Mean Time	Standard Deviation
Loopback setup	0.18ms	135.5us	0.2529
Loopback teardown	0.1ms	73.9us	0.0739
Remote setup	6.5ms-8ms	8590us	13.52
Remote teardown	0.1ms	150.1us	0.1962

4.4.3 Analysis

For the remote setup overhead, just as we had predicted, it's about 1.5 times longer than the remote RTT we measured before. The additional overhead might due to some other operations involved in the setup time, such as creating the receiving window for example. The local loopback setup overhead is also as we have expected, about 1.5 time longer than the RTT in loopback interface.

The connection teardown overhead is smaller then the setup overhead. This result can be explained by when tearing down the connection, rather than using 3 or 4 handshakes to close the connection as we described before, the client or server may just send one FIN packet and move on without waiting for any ACK packet come back. With the small standard derivation values, we believe our measurement is accurate.

5. *File System*

5.1 File cache size

5.1.1 Methodology

In this experiment, we will try to measure the system's maximum file cache size by reading the same big file for different length to explore the file cache effect.

Before we start exploring the file cache effect, we will need to first read the experimenting file and place it into the main memory. After that, when we try to read the data block from the same file again, if the file is already cached in memory then it will be read directly from it instead of going through the file system, which would be much faster and exhibit file cache effect.

However, when the file size exceeds the maximum file cache limitation, the entire file cannot be put into the main memory completely so the following read will cause cache misses and need to read from the disk, which will increase the data block read time significantly. So when we observe this dramatic change in data block read time, then it means that the experimenting file size is greater or equal to the file cache size for the testing machine. After that we can decrease the file size in finer granularity to seek the precise maximum file cache boundaries.

We performed the experiment as following: The maximum file cache size is measured by reading different size of data ranging from 512MB to 4GB from a 4GB file. For each experiment, we will perform the first read to make sure file's data get put into the main memory. Then we start the cpu clock measurement and read the same file again calculating the average reading time per data block.

There is one more thing we'd like to address, we choose to read the file backward instead from the beginning. The reason for reading the file in reverse order is to avoid prefetching the disk blocks.

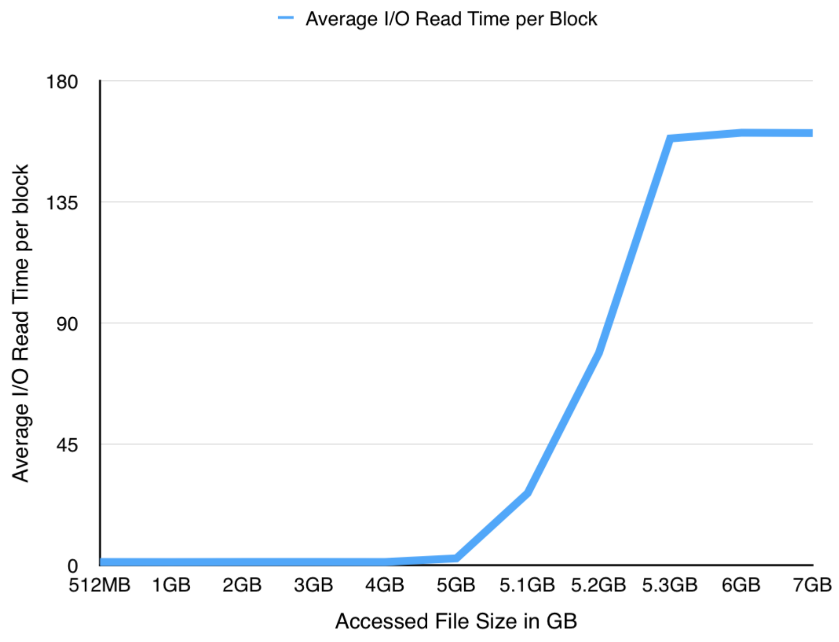
Prediction

The testing machine has 8GB main memory, since any unused RAM can be potentially used for file cache, we can predict the maximum file cache size by subtracting the memory space taken by necessary OS process's. By looking at the Mac OS's activity monitor, there are about 2GB of memory space used by OS and other user level processes, plus about 1GB of wired memory and 1GB of compressed memory. So we can predict that the available file cache size would be around 4-5GB.

Test Result

Here is the test result we obtained by reading different sizes of data from the same 8GB file ranging from 512MB to 8GB

File Size(MB)	Mean Block Read Time	Standard deviation
0.5GB	1.26us	0.083
1GB	1.23us	0.036
2GB	1.27us	0.058
3GB	1.275us	0.123
4GB	1.248us	N/A
5GB	2.60us	N/A
5.1GB	26.81us	N/A
5.2GB	78.89us	N/A
5.3GB	158.58us	N/A
6GB	160.72us	N/A
7GB	160.60us	N/A



Analysis

From the result graph, we can see that before the file size reaches 5GB, the data reading time per block is quite stable and small. Then there is a sudden steep increase after that, around 5.1GB. Then it stabilised around 5.3GB. Thus we conclude that the file cache size is around 5.2GB - 5.3GB. For the file size between 5GB and 5.3GB, we think the read time is a result of mix of file cache read and disk read. After 5.3GB, the read time becomes stable again which indicates that all the file cache has been used and we are now reading from disk.

Reference:

- https://developer.apple.com/library/mac/documentation/Performance/Conceptual/ManagingMemory/

5.2 File Read Time

5.2.1 Methodology

In this section, we will measure the file read time without the file cache effects. In order to do that, after we use the open() system call to acquire the file descriptor, we use the fcntl() call and set the F_NOCACHE flag to disable the memory cache. Before running each single read test we use "purge" command to clear out file cache. Thus the ReadTimePerBloack equals to the (total time)[(File size)(Block Size)].

For the sequential access, we read the file from the beginning to the end and calculate the average block reading time. We test on different file sizes, ranging from 2MB to 128MB. And we repeat the experiments for 10 times for each file size to calculate the average read time.

For the random access time, we will generate a list of random number between 0 and the number of data blocks the file has. Then we can iterate through this list and use lseek() to locate to that random file block and perform the read operation. We also repeat the experiments for 10 times for random access pattern.

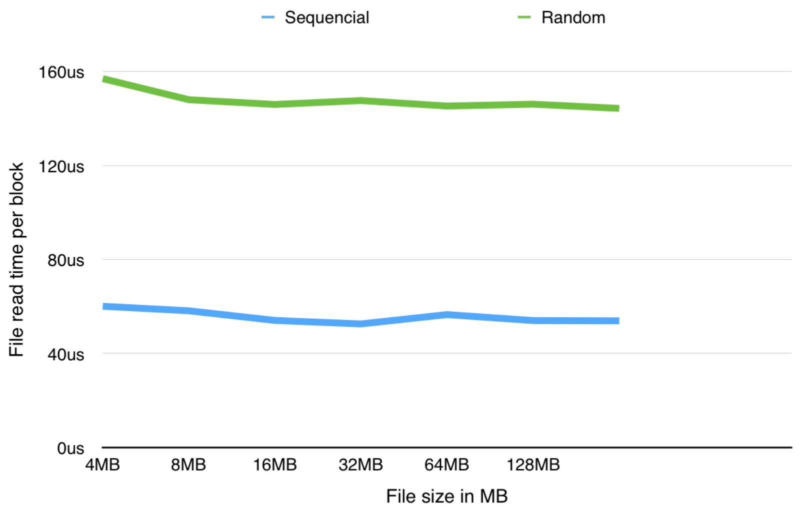
5.2.2 Prediction

Sequential access: Since our testing machine is using SSD as storage medium, the read latency time can be much more lower than the traditional hard disk drive since the data can be read directly from any locations without any mechanical operations involved. So most of the overhead would be from data access time. A reasonable prediction for that would be lower then 100us according to some SSD spec.

Random access: For the random file access, there is a additional part of overhead which would be seeking the target data block. Typical SSDs will have a seek time between 0.08 and 0.16ms. So a safe prediction for random access for a data block would be around 0.18-0.26ms regardless the accessing file size.

5.2.3 Test Result

File size(MB)	Sequential read(us)	Random read(us)	Standard deviation
2MB	60.07	157.56	3.21
4MB	58.14	148.84	3.478
8MB	54.04	146.00	1.697
16MB	52.55	147.67	1.675
32MB	56.52	145.32	3.18
64MB	53.98	146.12	2.11
128MB	53.86	144.32	2.76



5.2.4 Analysis

Sequential Access: The measured values are stable and matched with our prediction. As the file size grown from 2MB to 128MB, the sequential access time doesn't change too much. Because we have already prevent the main memory cache and we only access one block each read. Thus all the accesses are directed to the disk, thus why the outcomes for different file size are quite stable. At the same time, the deviation is low, which make us believe our method is correct.

Random Access: From the result we can see the random access time for different file size are stable but higher than the sequential access time. We think this is due to the effect of prefetching of sequential access and the overhead of lseek(), since basically both random access time and sequential access time are at the same level. For larger files, it is more possible the file is more fragmented. However, since we are doing random access, we believe this fragmentation effect is amortized through all the accesses. As we can see, the deviation is low, thus our method is accurate.

References:
http://arstechnica.com/information-technology/2012/06/inside-the-ssd-revolution-how-solid-state-disks-really-work/
http://stackoverflow.com/questions/2299402/how-does-one-do-raw-io-on-mac-os-x-ie-equivalent-to-linuxs-o-direct-flag
http://www8.hp.com/h20195/v2/GetHTML.aspx?docname=c04201478
http://www.anandtech.com/show/2829/22

5.3 Remote file read

5.3.1 Methodology

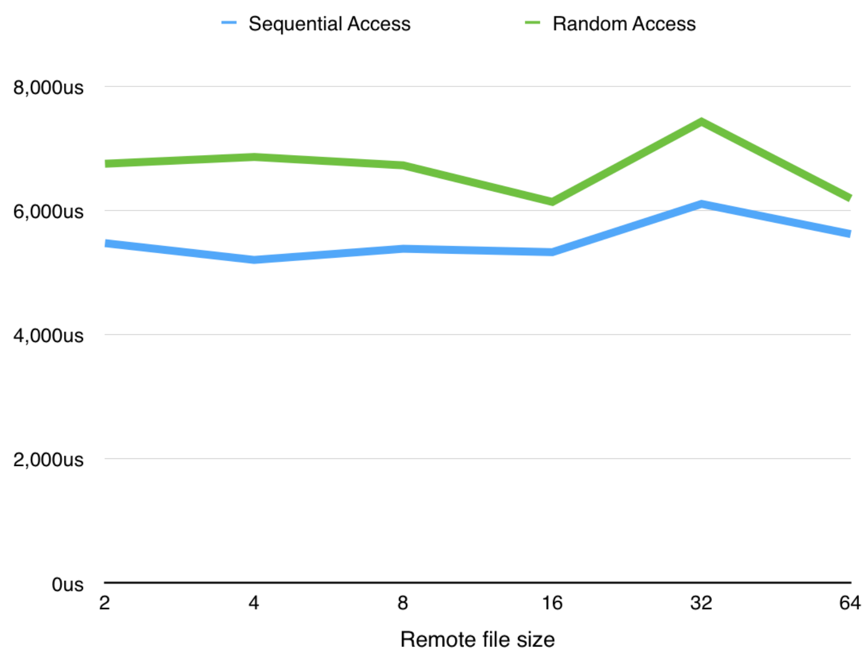
In this section, we use one Macbook Pro as a NFS server, and another Macbook Pro as the client. The server will share a file in the LAN, then we use the client to read the file via TCP connection. Before we start the experiment, we will use "Purge" command to clear the file cache on the client side. same as our previous experiment we use fcntl() and F_NONCACH in order to avoid cache effect.

5.3.2 Prediction

Because we have already measured the local file sequential and random access. The only difference here is we need to take additional network overhead into consideration. Besides, we can do this prediction based on the peak bandwidth we have measured before. Thus we guess the network overhead is around 4KB / 4.36 = 895.93 us. Based on all these estimations above, and our RTT estimation we have before, we think the overall sequential access time will be around 6ms and random access time will be 7ms.

5.3.3 Result

File Size(MB)	Sequential Read(ms)	Standard Deviation	Random Access Read(ms)	Standard Deviati
2	5.473	0.798	6.752	0.724
4	5.203	1.124	6.861	0.393
8	5.383	1.631	6.727	0.740
16	5.328	0.746	6.137	0.514
32	6.105	0.876	7.430	0.641
64	5.619	0.615	6.192	0.368



5.3.4 Analysis

From the test results, we can see that both the sequential access time and random time are also pretty close to our prediction. Plus, we notice that the difference between sequential and random access time in remote file access is lesser then what we have in the local setup. We think this might because the network overhead plays a more dominant rule in the total time, which makes the sequential and random access's different quite trivial.

As for this relatively huge block access time compared to our local setup, we think there might be several reasons for that. First, each layer in the network model will add different sizes of header to the data we try to send, so there is a larger data size overhead. Second, since we use WIFI to connect to the server, under some network contentions, the data transfering will take more time than our expectation. For different file sizes, local read overhead per block keeps constant, and since there is more added overhead from the network penalty, so the overall access time is much more longer than what we have in the local environment.

Finally, as we can see, the standard deviation is quite small compared with the access time, so the test result is pretty solid.

5.4 Contention

5.4.1 Methodology

For this part, we want to measure the file read contention when multiple processes simultaneously read different files in the same file system.

In order to conduct this experiment, we run multiple processes in the same time and each one of them will keep reading a different 64MB file on the same disk. Then we have another testing process that perform sequential or random read on another different file. For all processes, same as our previous experiment, we use fcntl() call to disable file cache.

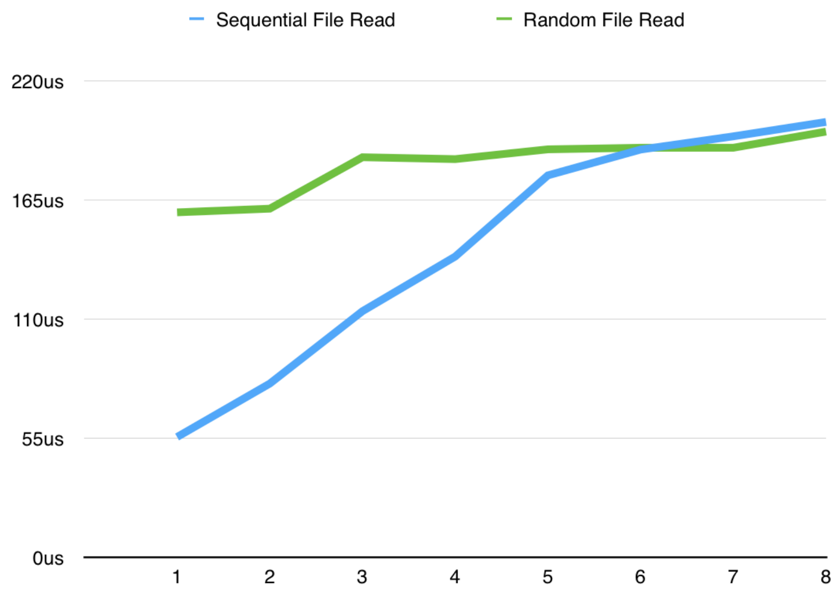
For both sequential read and random read, we measure the file block read time against how many processes are simultaneously reading the file. For each different process number, we conduct experiments for 10 times. And measure the mean and standard deviation value.

5.4.2 Prediction

For the sequential and random access read, with each additional process performing the file read operation, the file read contention will increase, as the result, the average read time per file block will increase proportionally to the number of processes that performing read operation in the same time. As for the sequential file access for every additional process, we predict there will be around 20 - 30 us's additional overhead due to context switch overhead and flush prefetched data blocks. As for the random file access, we predict the overhead won't be that significant, might be around 10 - 20 us since the data prefetching penalty won't be that big compared to sequential file access.

5.4.3 Test Result

Processes num	Sequential read(us)	Standard Deviation	Random access read(us)	Standard Deviation
1	55.54	1.60	159.233	4.59
2	80.20	1.97	160.945	4.49
3	113.62	2.20	184.679	6.32
4	138.835	2.70	183.809	5.20
5	176.311	3.51	188.33	6.38
6	188.227	5.57	189.051	4.82
7	194.329	13.30	189.065	8.31
8	200.962	11.83	196.508	4.08



5.4.4 Analysis

Sequential Access: From the result we can see that the file block access time increases as the number of processes that are simultaneously performing read operation increase. We'd like to point out that when there is only one process, for both sequential and random access the measured time matches our previous test result. The reason why the access time is increasing is due to the effect of prefetching decreases as the number of processes increases. When there is only one process, the prefetching data blocks can be stored in the main memory. When there are more processes and they are reading different files, the newly read data blocks by the current running processes will flush out data blocks read by previous processes. Thus when the previous process resumes, it cannot read its prefetched data blocks but have to do it again and flushes out other processes' prefetched data blocks. When the number of processes is getting bigger and bigger, the prefetching effect is gradually undermined, thus we can see a stable time reading after a certain point.

Random Access: For the random access since there is no prefetching effect, we can see the random access time is bigger than sequential access time at the very beginning, and gradually increases as there are more processes. When the processes number is big enough, we can see random access time becomes no bigger than the sequential access time, which we believe it is due to the sequential read can no longer benefit from the prefetching effect. As our test result in the previous section, the context switch time is about 100+us, and the random data block read time is about 150+us, which makes us believe there is only a really small chance that context switch happens during block read, thus although we are increasing the number of processes but the average data block reading time won't be affected too much.