CSE221 Report

System Measurement Project

Chun-Tse Shao A53049393

Hsin-Kai Wang A53046103

December 14, 2014

Contents

1. Introduction

In this project, we aim to build a solid, portable, and precise benchmark for Windows System. The Benchmark includes some measurement functions, which is listed below.

- Measurement and procedure / system call overhead
- Context switch overhead
- Memory RAM access time / bandwidth
- Page fault service time
- Network round trip time / peak bandwidth / connection overhead
- Size of file cache
- File / remote file read time
- Contention

We plan use Asus N550JV as our target machine. The OS on the machine is Windows, and we believe that there are abundant APIs on Windows, which are able to improve the precision to our project. Moreover, we will develop our benchmark in C, and we will use Microsoft Visual Studio Express 2013 RC as our development environment and compiler.

Since we are group of two, we plan to separate these benchmark functions to two parts. Part one is for CPU, scheduling, OS services and Memory, and Part two is for Networking and file system. In our plan, Chun-Tse will compose part one, and Hsin-Kai will do the second part. In the processing of developing our benchmark, we will compile and test our code on the target machine with designate compiler simultaneously. Since we believe we must spend at least three hours to every benchmark function, and we need extra time for composing report, we estimate we will spend 100 hours on this project.

2. Machine Description

Asus N550JV	
Processor	Intel® Core™ i7 4700HQ at 2.40GHz, 4 cores, 8 Threads
L1 D-Cache	4*32 KBytes 8-way set associative, 64byte line size
L1 I-Cache	4*32 KBytes 8-way set associative, 64byte line size
L2 Cache	4*256 KBytes 8-way set associative, 64byte line size
L3 Cache	6 MBytes 12-way set associative, 64byte line size
Memory Bus	Type: DDR3 Speed: 798.1 MHz Total Width: 64 bits Data Width: 64 bits
IO Bus	6.4 GB/s.
Ram	2*8192MB DDR3 (800MHz)
Hard drive	128GB SSD
Wireless Network	Integrated 802.11 b/g/n

3. CPU, Scheduling and OS Services

3.1 Measurement overhead

- **Methodology**

  The Time Stamp Counter (TSC) is a 64-bit register present on all x86 processors since the Pentium. It counts the number of cycles since reset. In this experiment, we used the RDTSC instruction both as a tool to measure elapsed time and as the object that is being measured, since it is essentially a read from the time stamp register of the machine. For the first loop, one register read instruction is executed in each iteration. On the other hand, the second loop includes two register.

  Read instructions which are executed per iteration. We can obtain a clean overhead time for reading time by subtracting the executing duration of the second loop with that of the second loop.

  We are reporting the time in the number of cycles the CPU takes to execute the designated tasks, instead of the time it takes because of the variation of CPU speed on different machines. For our own result, we can simply #define CLOCKS_PER_SEC 3292140000 to divide the cycles and get the result in second.

- **Result**

Empty Loop Time (cycles)	5.1
Read Time (cycles)	5.64
Pure Read Time (cycles)	0.54
Pure Read Time (ns)	0.164
1/(IO Bus Speed)	0.156ns or 0.5 cycle
Software Overhead (ns)	0.008

- **Discussion**

  Because the variable we are going to read have to be in the cache first, we make some previous call before the implementation. Besides, the function __rdtsc we use for measuring and the variables that save the time make additional overhead for the overhead of reading time. Therefore, we make two loops where one is for usage of recording only and the other including both the recording and variable which is read (without any other operation). The result is pretty close to the hardware I/O bus speed. The difference of actual result and I/O bus speed must be the software overhead. And also we find that a decrease in loop overhead when more iterations of the loop is executed. This might be explained by the existence of a hardware-based branch prediction mechanism in the machine.

3.2 Procedure call overhead

- **Methodology**

  Now, we read the time just before calling 8 function calls that accepts 0 to 7 int arguments. These functions would return immediately without any operation. We are doing so because the overhead varies according to the number of arguments passed in, and it is so because of the different number of push operation in the compiled code. We read the time again after these functions return and obtain 8 procedure call overheads, by calculating the elapsed time between each pair of two reads. These include creation and annihilation times of the additional function stacks in the memory. An overall loop is used to collect enough times of measurements for each function call. We recorded the averaged overhead for each function.

- **Result**

Base Hardware Performance	1000 Times Average(cycles)	1 Time With Pre-called(cycles)
0-Argument Procedure Call	46.029	180
1-Argument Procedure Call	46.743	153
2-Argument Procedure Call	46.788	747
3-Argument Procedure Call	46.782	267
4-Argument Procedure Call	48.204	294
5-Argument Procedure Call	47.544	294
6-Argument Procedure Call	72.879	285
7-Argument Procedure Call	47.682	894

- **Discussion**

  In our prediction, the overhead of function call is proportional to the number of arguments passed in. The experiment result shows that functions call time increase slowly with the number of arguments but sometime there is a peak in some number of the argument, so the increasing pattern is unknown. We assume that the optimization has been performed at CPU level for functions calls which are called several times. Therefore, we also make a call only for one time. Here, to make sure that procedure call is read into the memory, we make the call in advance before measurement. Zigzag behavior in the trend of the data could be explained with data alignment, that is, data access becomes more efficient when data are transmitted in blocks with some kinds of size.

3.3 System call overhead

- **Methodology**

  In our prediction, the overhead of function call is proportional to the number of arguments passed in. The experiment result shows that functions call time increase slowly with the number of arguments but sometime there is a peak in some number of the argument, so the increasing pattern is unknown. We assume that the optimization has been performed at CPU level for functions calls which are called several times. Therefore, we also make a call only for one time. Here, to make sure that procedure call is read into the memory, we make the call in advance before measurement. Zigzag behavior in the trend of the data could be explained with data alignment, that is, data access becomes more efficient when data are transmitted in blocks with some kinds of size.

- **Result**

GetCurrentProcessId() Function Call time	In Cycles	In Microseconds
Called First time	10257	3.1156
Called Once with Already Called Before	73	0.0222
Average 10 times with Called Before	9.6	0.0029
Average 100 times with Called Before	3.33	0.001
Average 1000 times with Called Before	3.057	0.0009

- **Discussion**

  Obviously, system call overheads are much larger than procedure call overheads in the first time (The time of procedure call in the first is about 300 to 400 cycles). This is the result of the additional time spent on switching the system between user mode and kernel mode. However, we have noticed that process IDs are highly likely to be cached when either get GetCurrentProcessId() are called second time. They return faster after calling it second time from 10257 to 73. As for GetCurrentProcessId(), Its speed can even rise to a level close to plain register read after being called a thousand or even just hundred times.

3.4 Task creation time

- **Methodology**

  Windows does not provide any mechanism that resembles process forking. All child processes are created as a new and different significantly from the parents in context and properties. To measure the process creation time, we called CreateProcess() on Windows. In addition, we terminate the child process once it has started running. Timestamps are recorded before the CreateProcess() and after the new process terminates. Therefore, the obtained process creation time is longer than the true process creation time. More accurate methods usually involve IPC mechanisms for sending data between processes. We reside to this method due to its simplicity.

  To measure the thread creation time on Windows, we create a thread using PsCreateSystemThread() to ensure it's a kernel thread. Timestamps are acquired before thread creation and within thread.

- **Result**

Task Creation Time	Average in 100 times(cycles)	First time without pre-called(cycles)
Process creation	1094270	3131890
Thread creation	283773	498463

- **Discussion**

  Thread can be regarded as a lightweight process. Hence, compared to process creation time, thread creation time is a lot smaller. According to the book, Operating System Concepts, which states that creating a process is about thirty times slower than creating a thread. However, we see that thread creation time only 4 to 6 times as larger overheads as process creation time. We think that window has been designed for multi-user and the thread has been developed more functionality and therefore the magnitude of memory and structure the thread has is different from the past right now. The thread is still quite light comparing to process, so creating a child process shouldn't be considered if a thread is sufficient to complete the job.

3.5 Context switch time

- **Methodology**

  A context switch is the process of storing and restoring the state (context) of a process so that execution can be resumed from the same point at a later time. We use blocking pipes as an efficient way to induce one process to yield to another for context switches between processes. When a writer process writes to a pipe and block, a reader process is best unblocked to read the data from the other side of the pipe. Therefore, context switch between processes can occur naturally and be timed. We can measure the total time a parent spent on writing to child through a pipe, and the child spent on writing back to parents. The measured time is then divided by two for two context switches. We create a child thread that is dedicated to busy-logging current timestamp onto a piece of shared memory between threads, with its parent spin-waiting for it to yield control after it has used up its allocated time-quantum. Once the child yields to its parent, the latter immediately terminates the former and read current time. Thus, the time difference between the two recorded timestamps is the context switch overhead.Draft of Memory Operation.

- **Result**

Context Switch Time	Average in 100 times(cycles)	First time without pre-called(cycles)
Process Context Switch (no pre-called)	62551	69562
Thread Context Switch	687.099	1169

- **Discussion**

  In Process context switch, process context switch time isn't much affected by whether there are other projects busy waiting for kernel to schedule or not. We established inter-process communication via pipes, hence induce a parent and a child process to perform context switch. The relationship of thread and process on context switch is close to the creation. Overhead of thread context switching is relatively small compared to process context switching. "Operating System Concepts" states that process context switch is about five times slower than that of thread. Our results shows it is about ten times, providing another reason not to create child processes but threads for parallel programming, for it is not only expensive in creation but also much heavier in context switching. However, context switch overhead is lesser than that of creation because creation involves allocating memory and resources. It is more expensive than saving and loading stored states. Hence, it seems to be an excellent machine for parallel programming nowadays for multi-user and lots of jobs running in the same time even though it has a heavy thread creation penalty.

4. Memory Operation

4.1 RAM Access Time

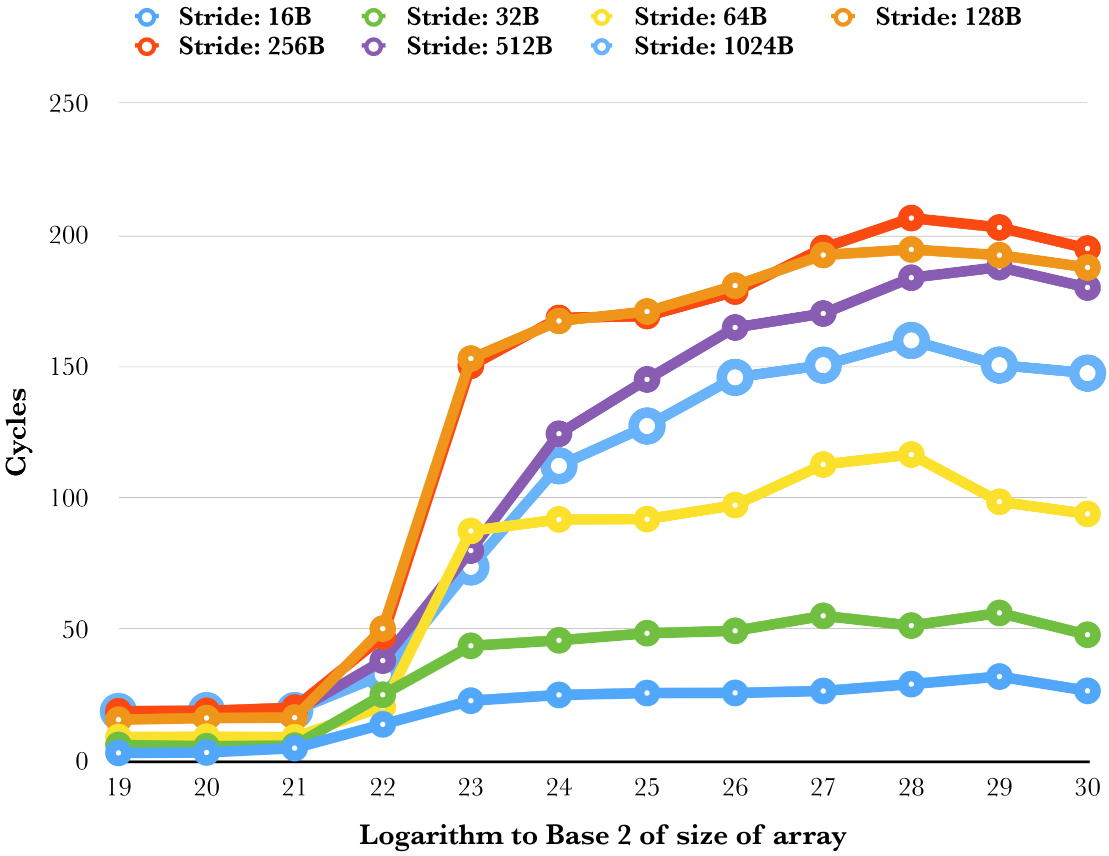
- **Methodology**

  In Operating system, RAM Access time indicates the latency of data arrival time from main memory to CPU. Our goal is measuring access time from different level of caches. In fact, RAM access time is not easy to be defined since the condition to start and end timing are not explicitly specified. To solve the problem, we plan to follow the "*lmbench*" paper[1] and use "*back-to-back-load*" latency. Back-to-back-load latency is the time that each load takes, assuming that the instructions before and after are also cache missing loads. Back-to-back loads may take longer than loads in a vacuum because memory stalls while MMU is busy even under optimization. We will only test back-to-back-load since this the general conception of latency.

  In this experiment, we will test the access time of L1 data cache, L2 data cache and main memory. The reason why we just test data cache is because we assume that the access latency of instruction cache and data cache is almost the same. Furthermore we plan to measure the latency of different data size in one access to memory. We plan to create an array with different size of an entry (or stride, which means the number of locations in memory between beginnings of successive array elements), from 1 kilobyte to 1 megabyte which are increasing in logarithm of base 2. The array will have thousands of entry which are stored in different location, and we will do accessing array of million times.

[1] Larry McVoy and Carl Staelin, lmbench: Portable Tools for Performance Analysis, Proc. of USENIX Annual Technical Conference, January 1996.

- **Result**



Since the disk of our machine is SSD, which is really fast, the result under array size 1 megabyte is too small to analysis. Our result of array size is from 512 kilobyte to 1 gigabyte.

- **Discussion**

Our result shows that Asus N550JV has multiple levels of cache. The access time is sharply increasing on several points, which means the access time to the next cache level. Approximately, the access time to L1 cache is ~5 cycles, to L2 cache is ~5 cycles (These two cases is generated while array size is smaller than 512KB), to L3 caches is ~25 cycles and to main memory is ~100 cycles.

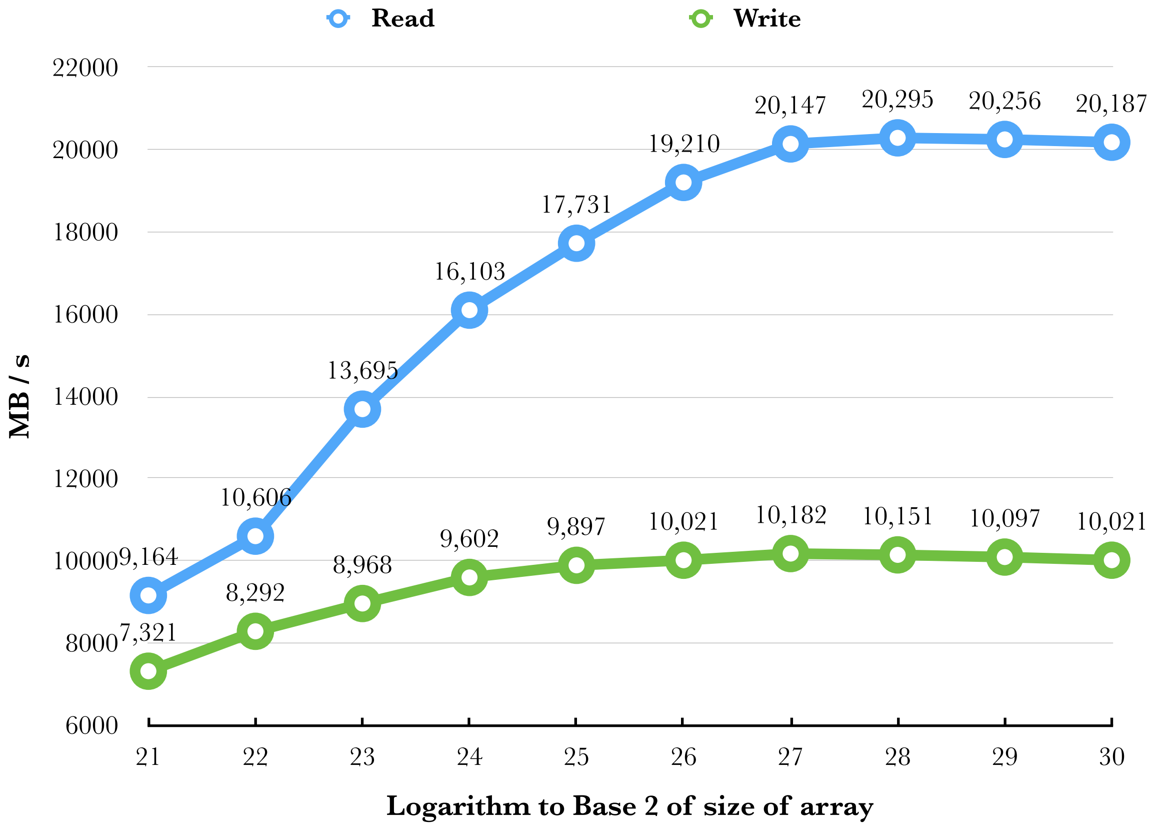
In our machine description we know cache line of Asus N550JV is 64 bytes. However, in our result, 64byte strides did not produce the highest latency for accessing memory. In fact 256 byte strides produce highest latency is our experiment, which means MMU may optimize memory access by fetching more than one cache line at the same time. The cpu of N550JV is Intel's i7 4700HQ, and in Intel's guideline paper for i7 cpu family, it says such the professor has 4 component hardware prefetchers. Such the prefetchers can prefetch the data like "streaming", so that we will think the machine fetch multiple cache line in the same time.

4.2 RAM Bandwidth

- **Methodology**

Memory bandwidth is the rate at which data can be read from or stored into a memory by a processor. In this experiment, we follow the the "*bcopy*" convention described in "*lmbench*" paper which is to counts the amount of data copied from one location in memory to another location per unit time. In other word, we have to measure the read and write bandwidth of the system. We need to perform the same measurement on arrays of different sizes to find out how large the read and write operations should be and use the largest measured bandwidth among all numbers as our result. In addition, we try to read only one character for each cache line to avoid cache-hitting in the same line which leaves available memory bandwidth wasted. And we need to use an unrolled loop of read/write to decrease the influence from instructions that control the loop such as pointer arithmetic and "end of loop" tests on each iteration. Finally, we should estimate the testing result to be lower than prediction because there may be some overhead such as hardware overhead like TLB and software overhead like resolving concurrency issues.

- **Result**



- **Discussion**

Our result shows that the highest ram bandwidth on writing and reading is generated when array size is around 256-megabyte. The highest throughput for reading is 20295 MB/s, and the highest throughput for writing is 10182 MB/s. In common sense, read is much faster than writing, which is also shown is our result.

In Intel's document, it indicates that the memory bandwidth of i7 4700HQ is 25.6-gigabyte per second, which is much higher than our result, which is almost 12 times than our result. We think this is because the multi-core system for i7 4700HQ, so that i7 4700HQ can execute several memory accesses simultaneously. Our result just test the RAM bandwidth for only one core. That is the reason why the total utilization on Intel's document is four times to our result.

4.3 Page Fault Service Time

- **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 in physical memory. There are three cases in page fault: major(hard), minor(soft) and invalid page fault. Major page fault is a mechanism used by operating system to increase the amount of program memory available on demand. These data may not be accessed at the beginning, and while a program want to use the data, hard page fault will be generated. Since we treat page fault as hard page fault in general, we only test hard page fault in the experiment.

  Since the memory space of a modern machine is usually extremely abundant, page fault is very rare. To the test, we use a brutal-force approach: filling memory space up and enforcing page fault to be generated. We plan to allocate thousands of arrays to fill up all the memory space. while memory space has been full occupied, operating system will start swapping out pages to disk for future usage, and the array which is allocated earlier may be the first victim. Once the memory is fully occupied, we begin walking through the first allocated array, and bring back the data that is just swapped out. So we can generate hard page fault and estimate the performance.

  On Visual Studio, we need to use the *Release* configuration to prevent access violation.

- **Result**

ASUS N550JV	Time(cycles)	Time(ms)
Page Fault Violation	1543374	0.4688

- **Discussion**

  We use the Windows tool "Performance Monitor" to obtain the actual number of page fault generated by our experiment. By the result we can find that the overhead of page fault is not very large. We think that is because the disk of our machine is SSD, so the page out mechanism will not consume much time. In traditional concept we seems the page out overhead is very heavy. However, while the speed of disk become faster, page fault will not be a serious overhead issue anymore.

5. Network

5.1 Round Trip Time

- **Methodology**

  First, we defined round trip time as the travel time for one byte from client through network to server, then server send acknowledgment character to client. To implement the experiment, we wrote two applications, client and server. We put these two applications to two different machines, one is ASUS N550JV, and another one is macbook pro with retina 2012. We set these two machines under the same local wifi router, which is TP-LINK TL-WR841N with 802.11n (300Mbps) standard supported, then use the client application to send one byte message to server. After server receive the one byte message, server acknowledgment character to client immediately. By doing this thousand of times, we can find average round trip time between these two machines. Furthermore, we also try local round trip time, which set client and server on ASUS N550JV, and test the round trip time.

- **Result**

ASUS N550JV	Round Trip Time(Local)	Round Trip Time(Remote)
Measure Time by *Ping*	~ 0.1 ms	~ 4.0 ms
Measure Time	1.231 ms	5.581 ms

- **Discussion**

  The round trip time in our experiment is 5.581 ms. We also try the function *ping*, which averagely takes about 4ms. That is faster than our experiment, we think that is because the *ping* is a primitive function from kernel and definitely run faster.

5.2 Peak Bandwidth

- **Methodology**

  For this experiment, we take approach similar to round trip time experiment. We make the client send a chunk of large data to server, and after server receive all of the chunk of data, server send acknowledgment character to client. Client will record the start time and the end time of sending chunk data, then we can find the bandwidth by dividing data chunk size to sending time. We choose the data size as 256MB, which is enough large to saturate the network bandwidth 300Mbps (37.5MB/s). After doing this for several times, we choose the largest one as our result for network peak bandwidth.

- **Result**

ASUS N550JV	Peak Bandwidth
Predict Time	37.5MB/s
Measure Time	5.628MB/ss

- **Discussion**

  Compare to the specification of 802.11n, which can provide 37.5MB/s wifi bandwidth, our result is much slower than the prediction. We believe it is because the unstable wifi signal and other network issues. Our summary is that peak bandwidth in network is hard to measure, and there are uncountable issues will affect experiment for bandwidth.

5.3 Connection Overhead

- **Methodology**

  We are going to separate connection overhead to two parts, setup part and shutdown part. After client and server sockets are initialized, we use client side to connect to server and set up, and send message to server, than shutdown the connection to client. We measure connection overhead on client, for the *connect()* and *shutdown()* functions repeatedly, and report the fastest one as our experiment result.

- **Result**

ASUS N550JV	Setup Overhead	Shutdown Overhead
Measure Time	0.532ms	0.158ms

- **Discussion**

  Compare to remote round trip time which only send onebyte message through wifi, the connection overhead is still small, just about 10%. However, compare to local round trip time it is almost about 50%. We think that because round trip time in the same machine do less in transmitting message, so that the connection overhead raise as an important issue. However, since most networking application transmit large chunks of data during one connection, connection overhead is less important in common case.

6. File System

6.1 Size of File Cache

- **Methodology**

  System file cache size grows and shrinks dynamically to t systems needs for an application ,and an OS will use a notable fraction of main memory (GBs) for the file system cache. On windows 8, this is option we can decide for minimum and maximum cache size but we leave it to be optimized to guarantee the performance. When the file is still in the cache, the reading time grows substantially lesser, and once the cache can't hold the whole file anymore, we can observe the evident overhead. In this way, we can make the program reads in files with increasing sizes, that is from small to large, it is expected to see an increase in overhead between certain file sizes.

- **Result**

Size of the File Being Accessed	Time(cycles)	Time(us)
1MB	39118.9	11.8825
2MB	38172.4	11.595
4MB	38182.7	11.5981
8MB	38416.1	11.669
16MB	39337.8	11.949
32MB	41525.4	12.6135
64MB	44431.4	13.4962
128MB	50645.4	15.3837
256MB	58671.3	17.8216
512MB	59318.9	18.0183
1024MB	73861	22.4356

- **Discussion**

  The graph shows that the curve grow slowly at the beginning and then there is a obviously increase before file size 64MB. And we can see that after that point, the timing does grow slowly any more even with some random. This means that a file with size larger than 64 MB is accessed directly from disk, and the file cache size is 64 MB at the time when we run the program.

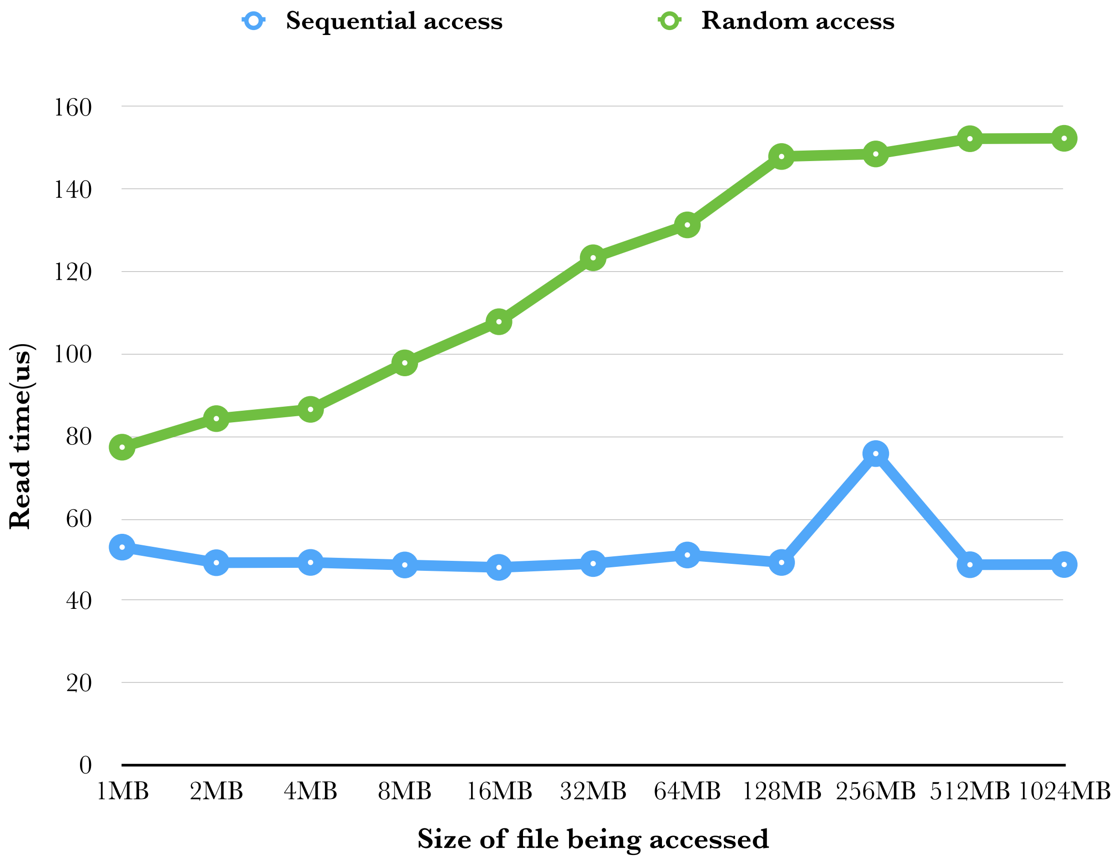
6.2 File Read Time

- **Methodology**

  On Windows 8, we can set FILE FLAG NO BUFFERING in ReadFile() system call to disable file caching in the memory. And now we create different sizes of many files. Second, we perform sequential and random access on each file and measure the overhead. In this section, all files are opened with file caching disabled. And we find out the physical and logical block size is 512 Bytes. So we design a block by block reading test method. We find that the size of the request is so small (512B) that it cannot fully utilize all the disk-IO bandwidth. But since this question is not for asking the max speed of the bandwidth, so we will just compare the relevant speed of sequential read and random read. Thus, the overheads reported, though not measured on raw device interfaces, are all bare overheads.

- **Result**

Size of the File Being Accessed	Sequential Access		Random Access	
	Read Time per Block(cycles/s)	Read Time per Block(us)	Read Time per Block(cycles/s)	Read Time per Block(us)
1MB	174601	53.0357	254649	77.3506
2MB	162143	49.2516	277529	84.3005
4MB	162368	49.3199	284940	86.5516
8MB	160342	48.7045	322174	97.8616
16MB	158392	48.1122	355122	107.8697
32MB	161438	49.0374	406345	123.4288
64MB	168321	51.1281	432662	131.4227
128MB	162345	49.3129	487430	148.0587
256MB	249517	75.7917	489474	148.6796
512MB	160534	48.7628	501671	152.3845
1024MB	160699	48.8129	502000	152.4844



- **Discussion**

  While HDDs are deeply affected by fragmentation, SSDs deliver consistent performance thanks to their use of integrated circuits instead of physical spinning platters – while rotating disks must wait for spindles motors, heads, and arms to physically locate data locations. SSDs can access any location with lightning speed. So there won't be the problem of fragments in our experiment. In the result, the characteristic of sequential and random file access isn't very different. For sequential accesses, when the file size grows, there is no increasing penalty. On the other hand, file size of random accesses grows with increasing time but not obvious. We think that it is because an non-volatile SSD does not drop when it comes to sequential data access pattern and SSDs's incredibly fast data access times.

  Therefore, both remote sequential and random file reading time does not grow as much as expected where the constant physical motion of the disk head casts tremendous overhead especially for random access patterns on disk.

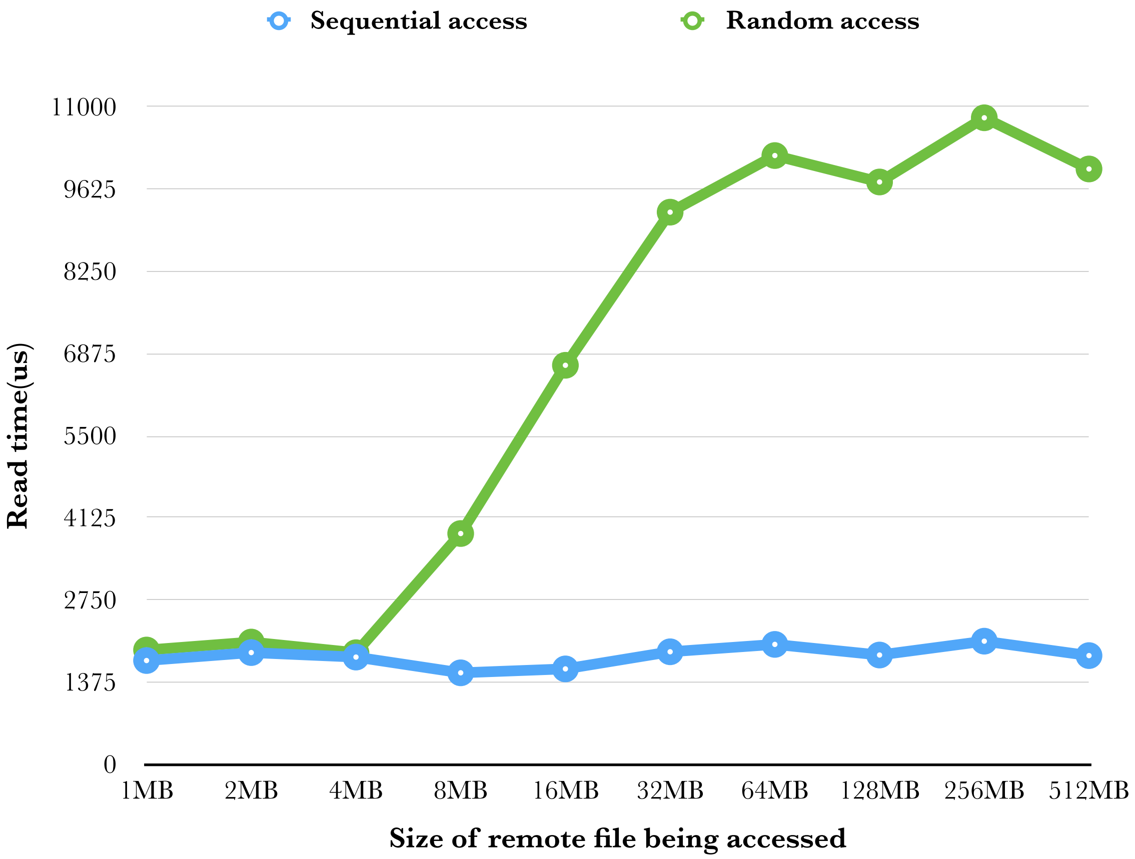
6.3 Remote File Read Time

- **Methodology**

  A remote file system on Windows is accessed through "Z:\" command which is the mounting point of a remote file system. The other steps is same as in local file read time to measure the performance. We consider the network overhead as the time taken for a remote read minus time taken for a local read. This overhead includes the time to process the protocol, bare network overhead, etc.

- **Result**

Size of the Remote Disk File Being Accessed	Sequential Access		Random Access	
	read time per block(cycles)	read time per block(us)	read time per block(cycles)	read time per block(us)
1MB	5732160	1,741.1653	6311780	1,917.2271
2MB	6170060	1,874.1791	6750740	2,050.5629
4MB	5922490	1,798.9788	6164350	1,872.4447
8MB	5056080	1,535.8035	12728700	3,866.3909
16MB	5271910	1,601.3626	21996800	6,681.6114
32MB	6215120	1,887.8663	30432200	9,243.8961
64MB	6621460	2,011.2936	33550300	10,191.0308
128MB	6035840	1,833.4093	32094500	9,748.826
256MB	6798490	2,065.0671	35632200	10,823.4158
512MB	6004250	1,823.8137	32811700	9,966.6782



- **Discussion**

  As clearly seen in the results, a remote file system adds to local penalties a uniform constant overhead to both sequential and random access file-reading. Interestingly, at the beginning in the case of remote random file access, there is no obvious increase in overhead as size of the files become larger. This behavior is very different from what we observed in local file access, where overheads grow slowly. We think that this effect can actually be accounted for by the storage device of our remote machine, another window 7 machine. In this example, the access device is disk which is different from the SSD used in local memory management. This observation is expected since for random accesses, the disk has schedule one disk seek for each access. Minimal disk seeking time always exists. When the file size is larger than 8 MB, significant delay can be perceived by users.

  As for network penalty, we compare remote sequential and local sequential access time before 8 MB, and the penalty is an almost close to each other. We think that this should be the network penalty. However, the network penalty varies with different network conditions, for instance, different signal quality and strength. But in this case, the result is reasonable and acceptable. Additionally, most file systems would implement specific optimization methods for better performance in reading or writing a file this way. Hence, this data access pattern nevertheless gives generally decent reading throughput, just like its physically accurate analogy. Therefore, our sequential read from files may not be physically sequential on disk at all. Our intended sequential access pattern is indeed an abstraction provided by the underlying file system.

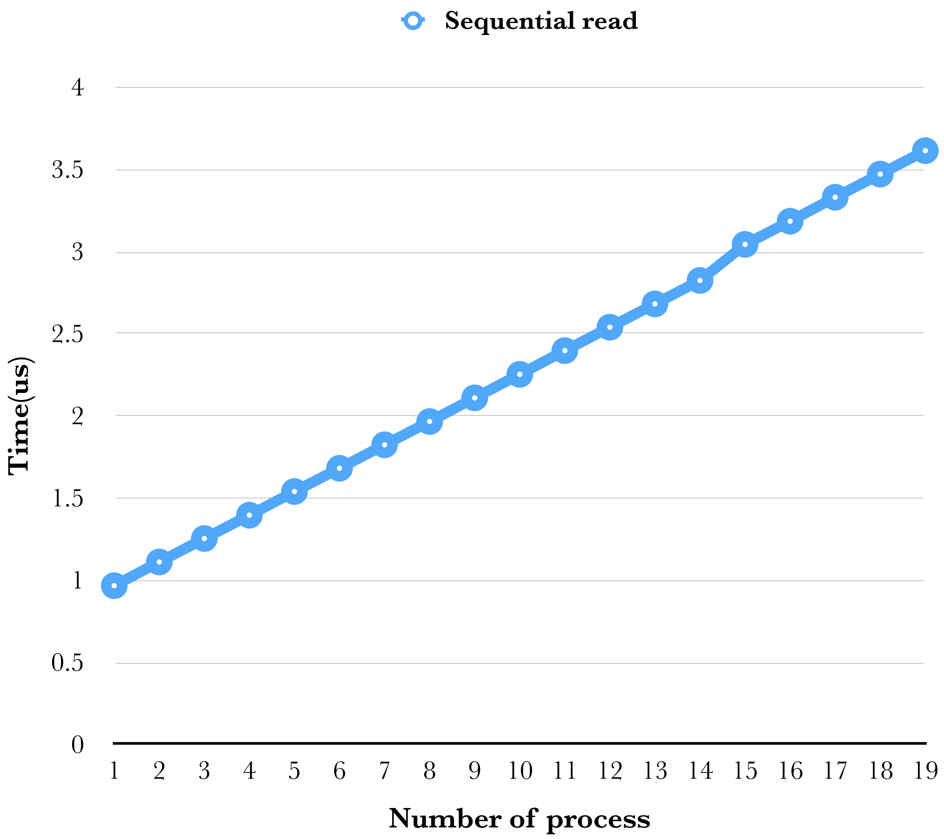
6.4 Contention

- **Methodology**

  We make sequentially read from different files by different numbers of processes. We followed the sequential way from the earlier implementation to read from different partitions. When multiple processes simultaneously request services from the file system and the disk, the tasks assigned to disk by different processes would be scheduled according to certain policy in order to maintain quality service. Nevertheless, degradation in system performance is virtually unavoidable and can be directly observed. Now, we create multiple child processes, each given a unique 32 MB file, from a parent process, and make them read their les persistently in order to measure the effect of contention among processes for disk service. In the way of process, parent process also has its own file, and would read a 512 bytes block of data from it after all other child processes have been assigned jobs to read from disk. The times which the parent process takes to finish reading a block from disk among various amounts of competing child processes are then measured and analyzed. We test the cases where 1 to 20 child processes would compete and interfere with the parent process' disk service request.

- **Result**

Number of process	Read time per block (cycle)	Read time per block (us)
1	3186.09	0.967
2	3660.75	1.111
3	4130.51	1.254
4	4598.13	1.396
5	5072.13	1.540
6	5538.31	1.682
7	6008.43	1.825
8	6474.47	1.966
9	6948.85	2.110
10	7420.11	2.253
11	7895.35	2.398
12	8363.53	2.540
13	8830.45	2.682
14	9300.57	2.825
15	10022.8	3.044
16	10487.2	3.185
17	10967.5	3.331
18	11431.1	3.472
19	11899.7	3.614



- **Discussion**

  It can be observed from our experiments that on Windows 8, the time it takes to read a block of data from disk grows linearly with the number of simultaneous disk requests. However, the slope of the growth is gentle, which indicates either the presence of some adequate disk scheduling strategies, or insufficient bandwidth utilization under small disk workloads. Although it is uncertain to what extent of process multitasking that this gentle linear behavior would pertain, the result actually suggests that certain level of concurrency can be beneficial in terms of overall performance of Windows NTFS file systems and hard drives.