Yiyun Fan - A53315911
yif007@eng.ucsd.edu

System Measurement Project

1. Introduction

The goal of this project is, as the title suggests, to measure the performance of the operation system and determine if it has served the hardware well. The target operation system for measurement is MacOS. A set of experiments will be performed to test the performance. The majority of the experiments are implemented in C for easy system measurement. I use gcc version 4.8.5 (GCC) to compile the code. A small portion of experiments (network) are performed with python; these scripts are runned in the terminal with a simple "python" command. Approximately 45 hours are spent on the project.

2. Machine Description

Processor: model, cycle time, cache sizes	Intel Corel i7, 4 cores, 2.2GHz -> cycle time $4.5 * 10^{-10} s$ L1 32KB, L2 256KB (per core), L3 6MB,
Memory bus	DDR3, 1600 MHz
I/O bus	AHCI Version 1.30 Supported
RAM size	8GB *2
Disk	Capacity: 250.79 GB SSD (no rpm info available) Cache size: Dynamically changing
Network card speed	AirPort Extreme (0x14E4, 0x152) 450 Mbps
Operating system (including version/release)	macOS Mojave 10.14.6

3. CPU Operations

3.1. Measurement overhead:

Methodology:

Yiyun Fan - A53315911
yif007@eng.ucsd.edu

There are mainly two parts in measuring measurement overhead: overhead of rdtsc() call itself and loop overhead. The method to find rdtsc() overhead is quite straightforward: do nothing between an end rdtsc() call and a start rdtsc() call. Since the interval between end and start is predicted to be small, I measured 100000 cumulation of such intervals to get the average result. As for loop overhead, I implement an empty loop that iterates 100000 times and measures the cycles taken between before and after the loop operation.

Expectation:
According to https://www.strchr.com/performance_measurements_with_rdtsc, the overhead for rdtsc() function is normally 150 ~ 200 clock cycles. So my guess is from 100 ~ 250. For the overhead of using a loop, I am guessing there is 1 cycle for the branch, 1 cycle for the index incrementation, and 2-3 extra cycles to access the memory. In total 4~5 cycles.

Result:

rdtsc() overhead

trials	1	2	3	4	5	6	7	8
Avg cycle	91.175960	93.118280	88.288700	87.712980	86.450940	88.833280	82.225660	89.596960

Loop overhead (per loop)

trials	1	2	3	4	5	6	7	8
Avg cycle	4.681060	4.629120	4.077340	4.079220	4.074840	4.258740	4.251080	4.278260

The result of measurement overhead is approximately 90 clock cycles. It is shorter than the reference suggests, so I suppose there have been some optimizations with the rdtsc() function since the referenced page was written. The loop overhead is similar to estimation

3.2. Procedure call overhead:

Methodology:

Yiyun Fan - A53315911
yif007@eng.ucsd.edu

In this experiment, I first implemented 8 empty functions, whose number of arguments ranges from 0 to 7. Then I call them iteratively in main function to measure the cycles required to call them.

Expectation:
I expect that for each more argument, the procedure call overhead would increase by one cycle or so. I expect a procedure call with no argument would be around 100 cycles.

Result:

Arg	0	1	2	3	4	5	6	7
Avg cycle for 20 tests	82.015	82.77	83.77	84.56	84.91	85.89	86.95	88.01

For a procedure with no arguments, the approximate overhead is 82 cycles. And for every more argument, the approximate increment is 0.5 to 1 cycle. I believe the reason is that for each more argument, the system will need one more cycle for assignment and access.

3.3. System call overhead:

Methodology:
In this experiment I used the getppid() method to measure system call cost. Some of my earlier experiments show that MacOS caches simple system calls such as getppid(). So instead of calling getppid() multiple times in one process, I create the processes multiple times for an accurate measurement.
Expectation:
I guess the system call will take 20000 cycles. It is a wild guess, but I believe it should be somewhat large - at least larger than the procedure call.

Result :

trials	1	2	3	4	5	6	7	8
cycles	41891	34160	29066	28336	29354	29628	34588	34244

The result turns out to be larger than my expectation, though not too far away. I believe the reason that system call costs much more than procedure call is that we have to cross protection domains to perform it, which could be very costly.

Yiyun Fan - A53315911
yif007@eng.ucsd.edu

3.4. Task creation time:

Methodology:
For process creation, I used the fork() function to fork a child process and calculated the cost between before calling fork() and after the child process is created. As for thread creation, I used the function pthread_creat() to create the thread and calculated the time before and after creation.

Expectation:
I expected the result to be large (>50000 cycles), as for each process creation we would need a fork().

Result:

trials	1	2	3	4	5	6	7	8
thread	54108.136	45399.544	47878.336	50663.644	45696.976	44775.256	47160.336	50932.844
process	1112459.628	1082860.136	1062448.468	1105073.896	1104421.380	1126427.470	1077373.228	1074174.120

The result seems to be even larger than I expected. Thread creation seems to be reasonable. And It also seems reasonable that a process creation requires more time than a thread creation. I think the main reason for high cost in process creation is that we have to cross protection domains, and the kernel needs to assign address space to new process created.

3.5. Context switch time:

Methodology:
Similar to the previous measurement, we first need to create a process with fork for process context switch measurement, or create a thread with pthread for thread context switch measurement. For both the process and the thread, I used pipi() to force a context switch from the parent process to the child process or from one thread to another.

Expectation:
I expect the context switch time would be smaller than creation time. Also the context switch time for threads would be smaller than that of the processes.

result:

Yiyun Fan - A53315911
yif007@eng.ucsd.edu

trials	1	2	3	4	5	6	7	8
thread	10006.87	10086.83	8894.07	8767.21	9650.31	9454.96	8947.57	9090.464
process	408861.18	454719.01	415866.03	398655.56	411544.37	405297.87	403828.06	399597.56

The context switch time for both thread and process is smaller than their respective creation time, and the switch time for thread is smaller than that of the process, so I think the result is somewhat reasonable.

For both process creation and context switch, I believe the main reason for their high cost is that we need to cross protection domains to perform these operations. And as the result from 3.2 suggests, system calls could be rather expensive themselves.

4. Memory

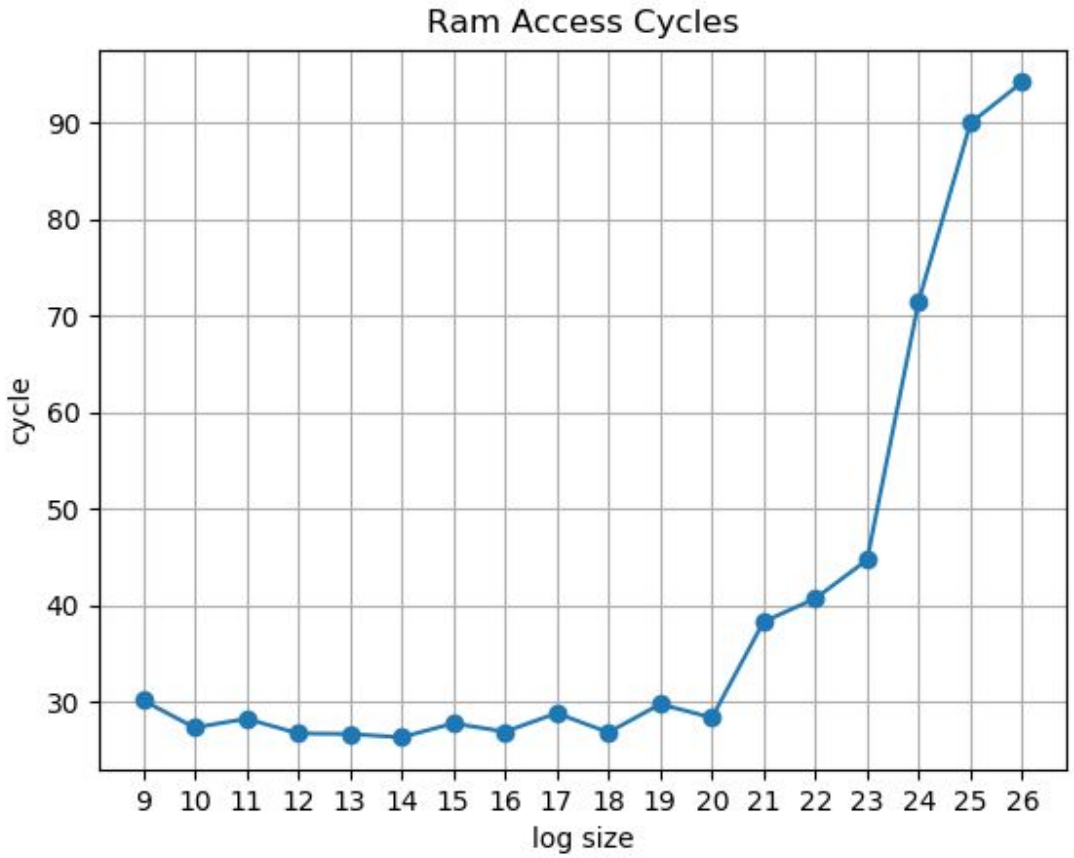
4.1 Ram access time

Methodology:
To test the ram access time, I build arrays of increasing size and test the average cycles required to access each array as the Imbench paper suggests.

Expectation:
I predict that L1 cache requires 10~15 cycles; L2 cache requires 25 cycles; L3 cache requires 50 cycles; and memory requires 100 cycles.

Result:

Yiyun Fan - A53315911
yif007@eng.ucsd.edu



There is a clear shift from 20 to 21, which I believe is the shift from L2 to L3. And then there is another shift at 22 to 23, which I believe to be the shift from L3 to memory. I think both shifts are pretty accurate, as the machine description suggests that L3 cache is 6MB - somewhere between 2^22 to 2^23. Size of each L2 cache is 256KB and there are 4 cores, so that total size is 2^20B. But there is also an abnormality: there is no clear transition between L1 to L2. One possible reason is that I didn't stop every application when conducting the test, so there were other programs using L1 cache.

4.2 Ram Bandwidth

Methodology:
I create an array of size 32 MB so that it is bigger than my L3 cache to ensure access to the memory. To reduce the effects of cache line prefetching, the array is not accessed consecutively - accessing array[j+1] does not follow accessing array[j]. Instead, I accessed the first item in the array followed by the last, and the second and the second to last, etc. I also implemented loop unrolling to optimize accuracy.

Yiyun Fan - A53315911
yif007@eng.ucsd.edu

Expectation:
The hardware performance expectation according to the article https://frankdenneman.nl/2015/02/19/memory-deep-dive-memory-subsystem-bandwidth/ for DDR3 1600 suggests that the peak transfer rate should be 12800 MB/s.

Result:

Trial	1	2	3	4	5
Read	7573.458301 MB/s	6891.879750 MB/s	7642.614721 MB/s	7686.636308 MB/s	7085.438889 MB/s
Write	6366.444822 MB/s	6387.486913 MB/s	6118.469659 MB/s	5965.409782 MB/s	6455.089346 MB/s

The results show the bandwidth is around 6200 MB/s for writing and 7300 MB/s for reading, slower than documented peak bandwidth for DDR3-1600Mhz (12000MB/s). There are a few possible reasons: 1) other programs are using memory bandwidth; 2) the experiment program does not read/write in peak bandwidth; 3) somehow the memory of my machine is no longer performing at its peak.

4.3 Page Fault

Methodology:
I use mmap as the hint suggests to perform this experiment. I record the time between before and after accessing the content from disk to get the page fault service time.

Expectation:
I expect that page fault would at least take 5000 cycles. But I do not expect it to be as large as accessing from traditional disks since the performance of SSD should be significantly better than a rotating disk. Also due to the lack of RPM data, it is hard to give a hardware performance prediction.

Yiyun Fan - A53315911
yif007@eng.ucsd.edu

Results:

	1	2	3	4	5
Cycles/byte	8231.79	8020.02	8528.93	9692.55	8549.12

By theory page fault is expensive, so it is reasonable enough that the result numbers are large. It is not only composed of disk access time, but also the transfer time needed to copy to memory.

As 3.1 suggests, the cycles needed to access a byte from memory should be approximated 100 cycles. In comparison, the average cycles of accessing a byte from disk in the occurrence of page fault is significantly higher - even though the disk is an SSD. The result makes optimization strategies such as prefetching very reasonable choices.

5. Network

5.1. Ping performance

Methodology:
I compare the result of running ping www.google.com to the result of using a python script to conduct a request.

Expectation:
I expect that the result of using ping would be around 10 ms, while the result of using python script would be around 60 ms.

Result:
Ping: Running the command line argument ping “www.google.com”, I got an average of 15.5798ms from 10 packets.
RTT with script: I used a python script to request google.com 10 times and the average time is 109.4ms.

Yiyun Fan - A53315911
yif007@eng.ucsd.edu

The result from the RTT experiment is much larger than the ping test and larger than I expected, considering that 60ms is the usual latency I get connecting with most commercial website servers. I suppose the main reason for the distinction between ping and the script is that “ping” is at kernel level while python script is run at user level.

5.2 Peak Bandwidth

Methodology:
I setup a server and a client with a python script that transmits data with TCP protocol. The client sends a designated amount of data, and calculates the time elapsed between establishing connection, sending the data, and receiving a receipt message from the server. The server side receives the data and sends a receipt afterwards. For the loopback result, I set the host IP 127.0.0.1 and the ran both client and server on the same machine. For the remote access, I prepared two machines in the same local network and ran the server side at one machine and the client side at another, where the host address was set to its IP address in the local network.

Prediction:
Hardware performance of my network card suggests that the loopback performance should reach 450 Mbps. As for the remote bandwidth, I expect that it would be limited by my Internet bandwidth and my network card performance.

Result:
Loopback: 383.08 MB/sec
Remote: 12.66 MB/sec
I am surprised at first with the loopback bandwidth, as it exceeds the network card limitation. Then I realise that the network card is in fact the WiFi network card, and transmitting data on the same machine would not require the WiFi interface. As for remote data transmission, 12.66 MB/sec seems to be a bit low. It indicates that the bottleneck is Internet bandwidth rather than the WiFi network card bandwidth. Another reason that might have caused the underperformance is that TCP protocol incurs a lot of overhead in establishing connection.

Yiyun Fan - A53315911
yif007@eng.ucsd.edu

5.3 Connection Overhead

Methodology:
Similar to 5.2 I setup a client side and a server side. For the setup overhead I calculate the time difference before and after establishing a connection. For the teardown overhead, I calculated the time difference between before and after closing the socket.

Expectation:
I expect the connection setup would be much more costly than teardown. Also remote setup would be much more costly than loopback setup considering that it would require more time to establish connection.

Result:
Loopback:

- Setup: 0.0485ms
- Teardown: 0.0077ms

Remote:

- Setup: 80.8ms
- Teardown: 0.156ms

The difference in both connection setup and teardown is significant between loopback connection and remote connection. I believe one possible reason is that loopback connection is actually particularly optimized. According to the document by Microsoft https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/hh997026(v%3Dws.11), some applications use loopback connection to implement inter procedure communication. It is possible that under such optimization, establishing and tearing down connection have been excessively simplified.

As for remote connection setup, the reason that it performs poorly could be the followings: 1) the TCP connection establishment is complicated and involves several acknowledgements, 2) the scripts are not optimized for better performance, 3) both the

client and the server are in user mode which makes it costly to use the network card hardware interface.

6. File System

6.1 Size of file cache

Methodology:

First I create a file that is large enough such as to display the impact of file caching. Then I read the file to memory for the first time by block. Finally I read the file again to calculate the time needed to read per block. To find the cache size, I increment the size of the file gradually.

Expectation:

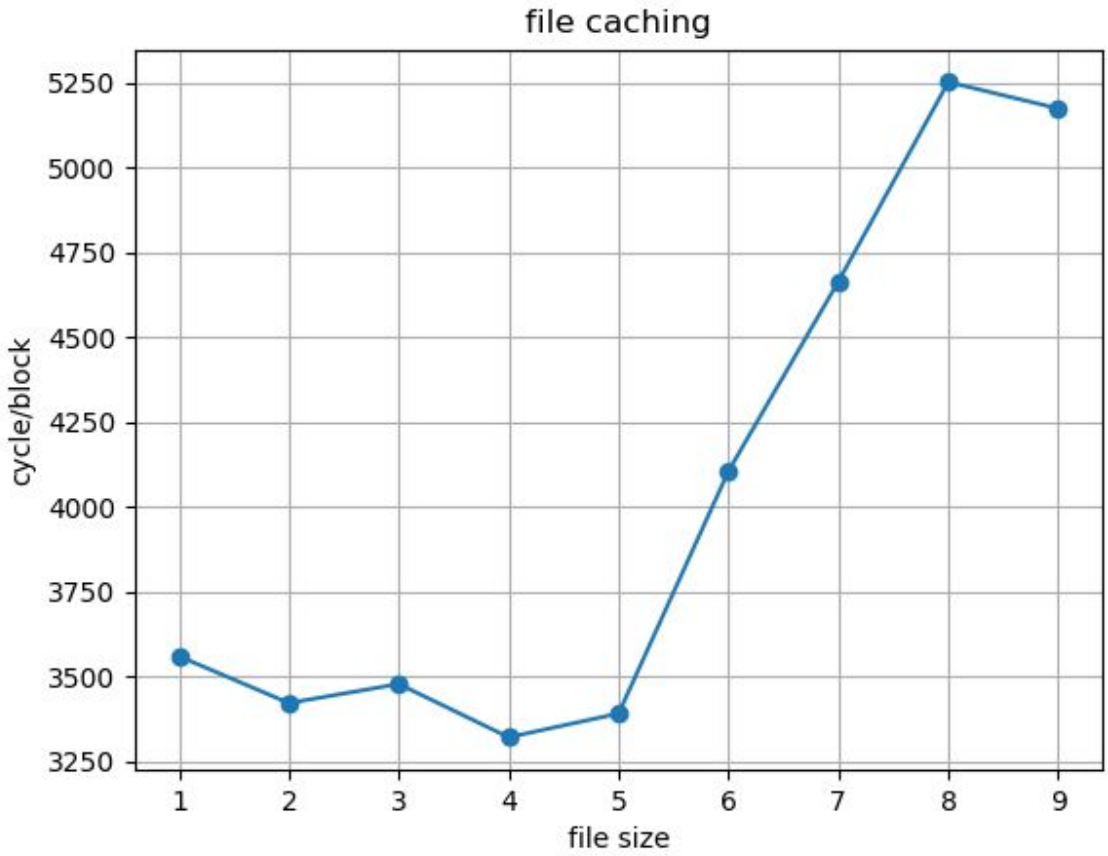
I expect the file cache size would be around 8 GB, which is half my total RAM size. There are other programs running the background so I do not expect it to take more than 10 GB.

result:

	1 GB	2GB	3GB	4 GB	5GB
cycles/block	3559.820679	3421.994749	3477.172656	3320.452019	3390.140398

6GB	7GB	8GB	9GB	* 7.5GB	* 8.5GB
4105.652463	4664.217434	5252.918371	5174.295216	3308.080799	5082.001263

Yiyun Fan - A53315911
yif007@eng.ucsd.edu



The result shows that when the file size is smaller than 5GB, the cycles to access data blocks are generally the same. Over 5GB, the cost required keeps increasing until it reaches around 8 GB. I think the memory used for file caching is around 5 GB, which explains why the cost keeps increasing after 5 GB. I also think there is "SSD caching" happening after 8 GB. Note that there is also an outlier which happens at 7.5GB. But I do not include it in the graph because I didn't perform the experiment after 7 GB. Instead it was done immediately after the 9GB experiment. The possible reason for such drop in cost is that the operating system considers the application requires a lot of file access and therefore grants more cache size to it.

6.2 File read time:

Methodology:

Yiyun Fan - A53315911
yif007@eng.ucsd.edu

For this experiment, I first create a new file. Then I use a file descriptor to read the file and to point to its exact locations. To avoid the effect of caching, I use the F_NOCACHE flag to eliminate the cache effects. For the sequential access, I access blocks by its location. As for random access, I use a rand seed to determine the location I would access next.

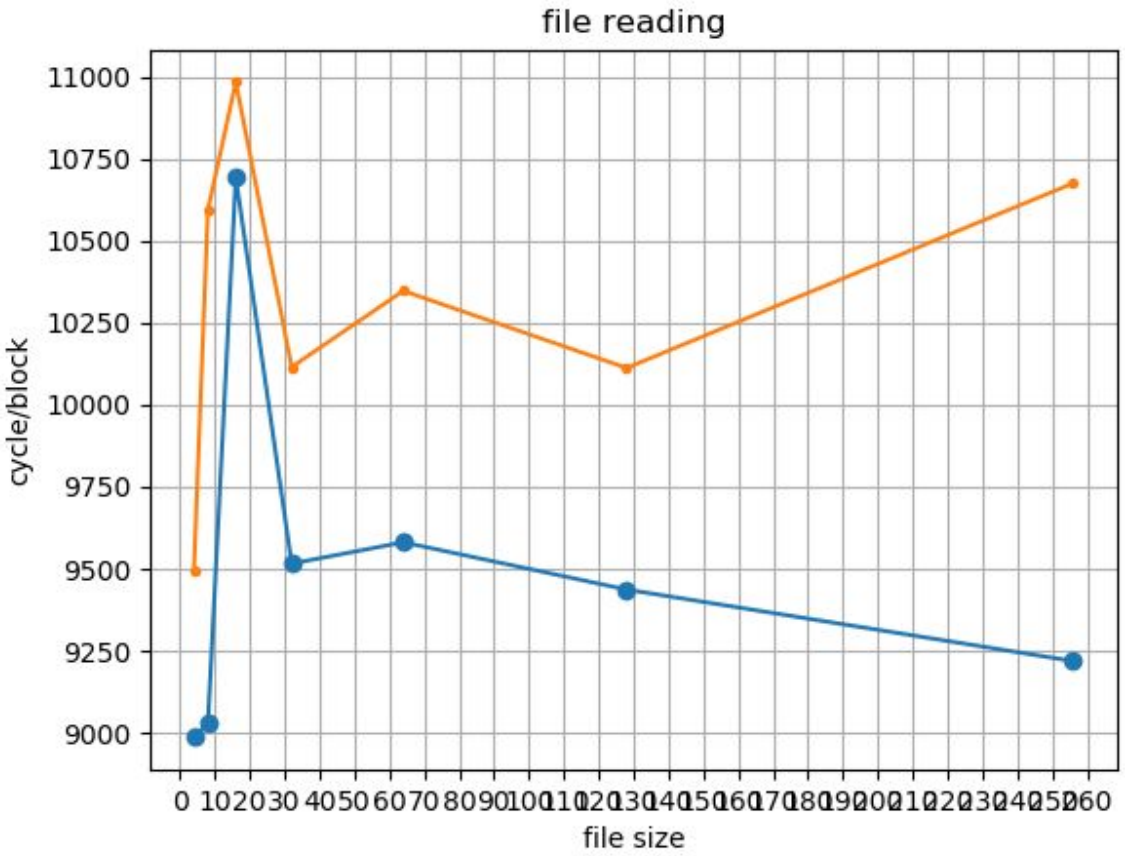
Prediction:

I predict that a sequential access would be around 7000 cycles and a random access would be around 10000 cycles. The prediction is made with the consideration of results from the previous sector in file caching.

result:

	4MB	8MB	16MB	32MB	64MB	128MB	256MB
seq	8987.7	9028.9	10693.9	9516.14	9581.15	9437.9	9219.53
rand	9497.9	10594.1	10983.5	10115.98	10347.6	10112.05	10674.1

Yiyun Fan - A53315911
yif007@eng.ucsd.edu



The result is not too far from my prediction. The cost is larger than the result from the previous sector, suggesting that file caching effect has been largely eliminated. Random access time is slightly larger than sequential access time. The reason could be: 1) sequential access is not actually “sequential”, as data blocks may not be put in neighbouring positions on disk; 2) unlike traditional rotating disks SSD does not need to move the pin around for data access which makes random access less costly.

There is a spike in the graph at 8MB file size. But I don’t think file size is a significant factor in this case because: 1) it could be the statistical errors in the experiment 2) they fluctuate around the mean value.

Yiyun Fan - A53315911
yif007@eng.ucsd.edu

6.3 Remote file read time:

Methodology:

For this experiment I tried the method of mounting onto an actual remote machine. However the experiment keeps failing because of access issues so I emulate the experiment with a client/server model instead. The server side creates a file, and waits for a message from the client side. The client side sends a message to the server side to indicate if it wishes a sequential or random access. Upon sending the message, the client side starts calculating time cost. Upon receiving the message from the client, the server side would access the file as requested and send back a message on completion. When the client receives the message, it stops the timer and reports the cycles needed per block.

Expectations:

I expect the result would be 10ms level and makes actual file access time marginal, considering the tests conducted in the network section.

Result:

The result in cycle is too large so I multiply it by 4.5 * 10^-10 for better interpretation

	4MB	8MB	16MB	32MB	64MB	128MB	256MB
Seq (ms)	5.293	6.184	5.837	5.919	6.012	7.301	6.554
Rand (ms)	5.300	6.092	6.201	6.074	6.133	7.523	6.732

I am not very confident with the accuracy of my result for two reasons: 1) it is an emulation of actual remote file access. 2) the experiment methodology does not closely simulate the case of file transmission. However I do think there are still some enlightenments from this experiment, that network and connection overhead (or penalty) is way larger than local file access time with today's computer, even in this simulation where client and server are essentially at the same machine.

Yiyun Fan - A53315911
yif007@eng.ucsd.edu

6.4 Contention:

Methodology:

For this test, I implement helper processes that would keep randomly access 4MB files. And a separate process runs to test the cycle / block like in 6.2.

Expectation:

I expect that when there is one more process running, the result would be slightly larger than the base case in 6.2. Then the cost increases as the number of processes increases.

Result:

processes	1	2	3	4	5
sequential	13866.18	15738.11	19631.74	23345.44	26741.28
random	13980.05	16946.41	19965.87	25892.53	28767.58

I am quite surprised that both the sequential and random file access time increase substantially when there are multiple processes running. Though it is not surprising that the amount of costs increases as the number of processes increases. The cost might have been induced by the CPU scheduling all the disk accesses. I also observe an interesting fact in this experiment, where the machine starts humming loudly after 3 processes running in the background. It shows in another way that the CPU is enduring to schedule for all the processes.

7. Summarize

* In some of the operations I am not certain how to find out its base hardware performance, so I include the possible impacting hardware in the brackets instead.

Operation	Base Hardware Performance	Estimated Software Overhead	Predicted Time	Measured Time
Measurement overhead	/	/	200 cycles	90 cycles

Yiyun Fan - A53315911
yif007@eng.ucsd.edu

(rdtsc())				
System call overhead	/ [CPU]	/ [switching to kernel mode]	20000 cycles	~30000 cycles
Procedure call	/ [CPU]	~1 more cycle per argument	100 cycles	~ 85 cycles
Process creation	/ [CPU]	Cross protection domains; assigning address space	50000 cycles	~1100000 cycles
Thread creation	/ [CPU]	Assigning address space in user mode	10000 cycles	~50000 cycles
Process Context Switch	/ [CPU]	Crossing protection domains	25000 cycles	~ 400000 cycles
Thread Context Switch	/ [CPU]		10000 cycles	~10000 cycles
Memory Bandwidth	12800MB/s	/	12800 MB/s	7300MB/s Read 6300MB/s Write
Page Fault	/	Searching in memory + accessing disk	5000 cycles/byte	8231.79 cycles/byte
RTT	[network bandwidth, CPU, network card bandwidth]	Crossing protection domain for RTT at user mode	60ms	109ms
Bandwidth	450Mbps	TCP connection overhead, crossing protection domains	50MB/s	12.66MB/s
Connection Overhead	/	TCP connection overhead	3ms for loopback, 60ms for remote	Loopback: 0.0485ms, 0.0077ms. Remote: 80.8ms 0.156ms

Yiyun Fan - A53315911
yif007@eng.ucsd.edu

File cache	Total memory 16 GB	/	8GB	5GB
File read time	[SSD disk] access time not sure	Disk access overhead	Seq: 7000 cycles/block Rand: 10000 cycles/block	9000 cycles/block 10000 cycles/block
Remote file read time	[Network Bandwidth]	All Network overhead and all file read overhead	10ms	5.3ms
Contention	[CPU]	Context switch overhead + file access overhead	10000 cycles/block	14000cycles/block ~ 27000 cycles/block