CSE 221 Project

Yixin Zou A53238025 Tao Li A53305875 Zhoulin Chen A53306003

1.Introduction

Code link:
https://github.com/zouyixin116/CSE221PJ.git

In building an operating system, it is vital to be able to determine the performance characteristics of the underlying hardware and analyse how these components have influence on the performance of an operating system. In this project, our goal is to use C++ language to design a set of experiments to a system to understand its performance. Various benchmarks will also be introduced to help us characterize the operating system. An estimated 120 hours will be spent on this project.

2.Machine Description

CPU Model	Dual-Core Intel Core i5
Processor Speed	3.1 GHz
Cycle Time	0.32 ns
L1 Cache	32 KB (Data) and 32 KB (Code)
L2 Cache	256 KB
L3 Cache	4 MB
Memory Bus	Size: 4 GB Type: LPDDR3 Speed: 2133 MHz

I/O Bus	SATA
RAM size	8 GB
Disk	Capacity: 251 GB TRIM Support: Yes Model: APPLE SSD AP0256J RPM, controller cache size
Instruction Set	64-bit
Networking Card Speed	Wifi Interface: AirPort Extreme (0x14E4, 0x171)
Operating System	macOS Catalina 10.15 (Oct 7, 2019)

Table 1. Machine Description

3. CPU Operations

3.1 Measurement Overhead

In order to compute the overhead of reading time, we use two rdtsc() with nothing between them. In this way, the difference is just the overhead. Result as below.

Iteration N = 1000000, overhead average =109.06.

Similarly, we put a loop between two rdtsc(), and get the difference deduct reading overhead as loop overhead. We guess that the loop overhead will be a linear function with iteration number N.Result as below.

Loop number N=1000000, loop overhead = 5666217.

3.2 Procedure call Overhead

To measure the overhead of procedure call, we target how overhead increases as the function parameters grow. We created 8 functions respectively with 0-7 integer parameters. These functions each were called for 10000 times to measure how they generate overhead. Finally, the average overhead was calculated for discussion.

A local procedure involves placing the calling parameters on the stack and executing some form of a call instruction to the address of the procedure. The procedure will then read the parameters from the stack, allocate stack space for local variables, do its work, place the return value in a register, readjust the stack pointer and then return to the top address on the stack. Based on the procedure, we assume that when more parameters are used, the procedure call overhead will increase accordingly. Our experiment results are shown as below:

# of P	0	1	2	3	4	5	6	7
Trial 1	29.67	28.99	29.58	31.33	31.79	31.46	32.10	32.15
Trial 2	26.81	26.01	25.88	27.76	27.06	25.72	27.76	29.32
Trial 3	27.63	26.43	26.56	27.76	29.44	29.66	30.18	29.78
Trial 4	28.08	25.65	27.98	31.26	33.95	32.67	41.99	29.33
Trial 5	24.89	24.45	24.68	31.17	32.93	26.55	23.43	24.42
Trial 6	22.56	22.95	23.08	23.48	23.21	23.45	23.02	23.07
Trial 7	23.56	23.66	24.35	23.72	23.28	23.61	52.17	29.00
Trial 8	27.03	26.42	26.56	26.75	28.51	26.03	24.29	24.47

Trial 9	24.45	24.45	22.57	23.10	23.05	22.98	24.18	23.04
Trial 10	23.51	24.19	23.63	23.87	26.27	23.52	23.51	22.92
Ave	25.82	25.32	25.49	27.02	27.95	26.57	30.26	26.75
std	2.34	1.75	2.24	3.39	4.08	3.53	9.69	3.49

Table 2. Procedure Call Overhead

Generally we can conclude from the table that when parameter number grows, the procedure call overhead increases. However, there also exists some irregularities when parameter number reaches 7, the overhead drops.

3.3 System call Overhead

When a process calls a system call, there will be a change from user mode to kernel mode. Current context of the process will be saved for return purpose. The program counter will jump to the address where the system call's kernel code is and then start to execute the code. There is the system call interface, which acts like the link to the system call operating system provided, in the middle. It will invoke the system call, return the status along with the return values.

Since there are more things to be done in a system call than a procedure call, we guess that it takes more time. We conduct the experiment by call the getppid() system call between two rdtsc(). Results are as below.

Iteration N = 1000000, average overhead = 1696.34.

3.4 Task creation time

In order to test the time consuming of task creation, we need to test the time of creating a process and a thread. We call rdtsc() twice before and after the procedure call, and calculate the time difference between these two time stamps. For process, we use fork() to produce a child process, and using wait(NULL) in parent process to make sure child process exit first, record the child process ending time stamp when return to parent

process. Then we could get the creation and running time of a process. For thread, we just create a new thread and let this thread join the main thread. In the function running in the new thread, for simplicity, we exit directly. Because we have already kept only one cpu and turned off the hardware multithread function, we could make sure that our process will have a high cpu rate during executing by using ***nice*** program in command line. Experiment results are as below.

With iteration N = 10000, average process creation time = 1231790, average thread creation time = 63538.2.

3.5 Context switch time

For context switching, we simulate process switching and thread switching in our program. The basic principle for time measuring is the same as experiments before, the biggest difference is the use of pipe. Because we need to measure the consuming time for context switch, so we must record the start time in one process/thread and record the end time in another process/thread. Referred to our course website project description, blocking pipe is a good method for us to perform internal process communication. Before creating a new process or thread, we first create an int array as pipe, then we first record the start time in main process/thread, we should store the end time by call rdtsc() in child process/thread after context switch and write it into the pipe. When the child process/thread finished, we could read the end time from the pipe, and get the time difference. In the whole process above, we must guarantee the synchronization of our program, so procedures such as wait(NULL) or pthread_join() are necessary. Experiment results are as below.

With iteration N = 10000, average process context switch overhead = 566710, standard deviation = 9719.17; Average thread context switch overhead = 7335.83, standard deviation = 934.72.

4. Memory Operations

4.1 RAM access time

In this part, our target is to measure memory access latency of main memory and L1,L2,L3 caches, then get the size of L1,L2,L3 caches as well. According to lmbench paper, memory access latency has four definitions: memory chip cycle time, processor-pins-to-memory-and- back time, load-in-a-vacuum time, and back-to-back-load time. Here we focus on back-to-back- load time, because it is the only measurement that may be easily measured from software and because we feel that it is what most software developers consider to be memory latency.

As stated in the lmbench paper, we set different sizes array(from 512B to 256MB) and visit them with various stride(from 8B to 256KB) to measure memory access time under different circumstances. By analysing these datas, we could infer L1,L2 and L3 caches size.

In our experiment, we still use the rdtsc() function mentioned before to measure the time difference before and after memory access action. With memory access methods offered in the paper, we use pointers to realize circular access of memory by simply assigning the address of the next memory accessed by process to each pointer variable in an array. For each experiment, we do about 1,000,000 loads and calculate the average consumption for loading each time.

Experiment results

We did experiments with array size from 4KB to 512MB, and stride size from 8B to 256KB. Because the size of an integer pointer is 8Byte in C++, so we start stride from 8B. Our experiments record each experiment's time difference and show it in the graph below. The x-axis represents the logarithm of array size, y-axis means how many cpu cycles we use for each specific experiment.

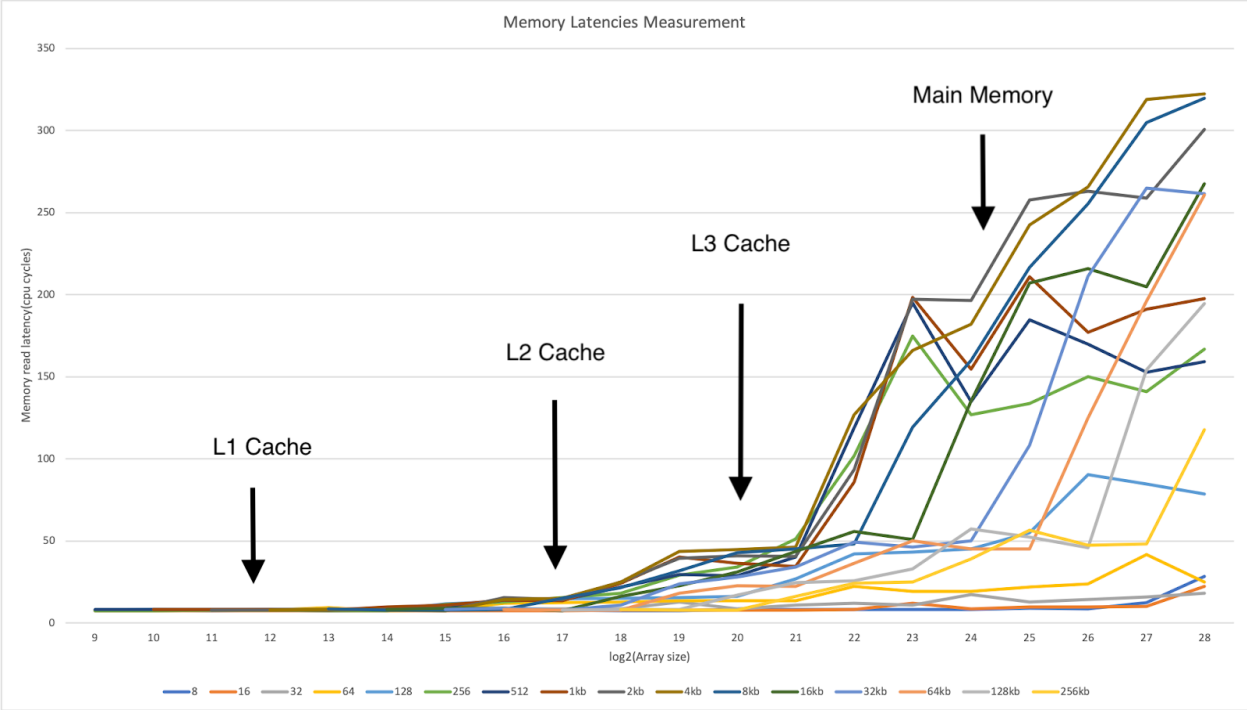

Figure 1. RAM Access Time

Analysis

According to the graph above, we could infer a lot of useful information. There exist three obvious turning points at 32KB-64KB, 256KB-512KB, and 2MB-4MB, from which we could infer our L1,L2 and L3 caches size. So we could have a reasonable guess that our L1 size is 32KB, L2 is 256KB, L3 is 2MB. Accessing an array with a smaller size than cache is definitely faster than an array the size of which is bigger than cache, because the latter one needs to be loaded into caches much more times. Compared with the actual data in the machine description part, our guesses are all proved correct.

Simultaneously, we also observed that when array size is quite large(eg. 512MB), the first significant leap lies between stride size of 64Bytes and 128Bytes. So we think it is very possible that the size of the cache line is 64Bytes. As stated in the paper: The cache line size can be derived by comparing curves and noticing which strides are faster than main memory times. The smallest stride that is the same as main memory speed is likely to be the cache line size because the strides that are faster than memory are getting more than one hit per cache line.

4.2 RAM bandwidth

4.2.1 Methodology

Memory bandwidth measures the amount of data that can be accessed in a given amount of time. In order to measure the bandwidth of the main memory, we referred to the Imbench paper's approach to implement measurement.

For memory reading, we used an unrolled loop that sums up a series of integers, and each time an exact size of the cache line is read from memory (16 integers, 64 Bytes). The loop is unrolled to reduce the for-loop overhead and amortize among the instructions. For memory writing, it is also measured by an unrolled loop which stores an integer into the array.

Since we need to cross the L1, L2 and L3 cache and do operations on the main memory, the array we operated on needs a size larger than the L3 cache (4MB). Then we can avoid cache hit. When we tried to write on the main memory, before we actually did the writing operation, the cache space was filled with a foo array to avoid writing into the cache space.

4.2.2 Base Hardware performance prediction

The machine we tested on has 8GB memory of LPDDR3 with a speed at 2133 MT/s, 64 bits width of memory bus, double data rate, which can achieve the maximum bandwidth of 1,066,000,000 clocks per second × 2 lines per clock × 64 bits per line × 2 interfaces = 272,800,000,000 (272.8 billion) bits per second (in bytes, 27,280 MB/s or 27.3 GB/s)

4.2.3 Results and Analysis

Base Hardware Performance estimation: 27.3GB/s
Observed Memory Read Bandwidth: 14.75 GB/s
Observed Memory Write Bandwidth: 18.31 GB/s

The results show that the actual read/write bandwidth is less than the estimated base hardware performance. Till now, we haven't figured out the reasonable explanations for the discrepancy and plan to explore the reason in the next step.

4.3 Page fault service time

4.3.1 Methodology

Page fault is a type of exception raised by computer hardware when a program accesses a memory page that is not currently mapped by the memory management unit (MMU) into the virtual address space of a process. The processor's MMU detects the page fault, while the exception handling software that handles page faults is the kernel. When handling a page fault, the operating system tries to make the required page accessible at the location in physical memory or terminates the program in cases of an illegal memory access. So several events happen in the Page fault process:

Hardware traps the kernel and current context is saved; the kernel notices a page fault and tries to find a page frame for it, if it cannot, run a replacement algorithm. If the frame to be replaced is dirty, write it to disk first. Then page table updated, the page it wants now acquired. Finally, restore the process context and let it continue.

In the experiment, we make use of the mmap, which is a POSIX-compliant Unix system call that maps files or devices into memory. It implements demand paging, because file contents are not read from disk directly and initially do not use physical RAM at all. We get a movie file which is about 3GB to do mmap, and set the access offset is large number 1e9+7. We use rdtsc() to compute the access time, and get the average of 100 times access.

4.3.2 Predictions

We make a prediction before we record the result. There are some important time cost parts, disk access, transfer time, context saving and switching time. According to the computer reference, we predict it should be around 600000.

4.3.3 Results and Analysis

The result is 753513.67. Since a page is 4KB, the time to access a byte is Totaltime / page size, which is 183. We then compute the time to access a byte in memory, which is 114.95. It is obvious that memory is much faster and that of disk.

5.Network

5.1 Remote Machine Description

CPU Model	Dual-Core Intel Core i5
Processor Speed	3.1 GHz
Cycle Time	0.32 ns
L1 Cache	32 KB (Data) and 32 KB (Code)
L2 Cache	256 KB
L3 Cache	4 MB
Memory Bus	Size: 4 GB Type: LPDDR3 Speed: 2133 MHz
I/O Bus	SATA
RAM size	8 GB
Disk	Capacity: 251 GB TRIM Support: Yes Model: APPLE SSD AP0256J RPM, controller cache size

Instruction Set	64-bit
Networking Card Speed	Wifi Interface: AirPort Extreme (0x14E4, 0x171)
Operating System	macOS Catalina 10.15 (Oct 7, 2019)

Table 3. Remote Machine Description

5.2 Round trip time

5.2.1 Methodology

In this part, we need to test the round trip time of loopback and remote machine. RTT could be tested based on ICMP or TCP. For ICMP we use ping in the shell, as for TCP, we use python program to measure the RTT.

Here we use two identical machines in the same local area network to test the remote machine performance. We set a server running on the remote machine, and run a client on the local. We record the time difference before client sending and after receiving the reply packets from the server, which represents the RTT form local to remote machine. Here we set our sending message 32 bytes, which is exactly the same as ICMP. In order to calculate the time difference more accurately, we run the process 100 times and get the average to compare with ICMP latency.

About the loopback, we just simply change the host ip to '127.0.0.1', and keep the other part unchanged.

5.2.2 Predictions

For localhost and remote machine, the rtt can be different. Because when a machine send a packet to its localhost, it does not need to send the packet into the internet. However, through the tcp/ip stack in the kernel, the packet will pass directly to the target process. So we predict the loopback rtt will be very small, maybe 0.1ms. Because the "ping"

operation is based on ICMP, while our tcp program uses socket, which is at a higher level. So we predict the time of ping will be shorter than our user-level program.
About the remote machine, because of the overhead of network latency, congestion, routing etc. Its overhead will be much higher than localhost. Conserving our tested machines are in the same local area network, we predict that the remote machine rtt will be about 10ms.

5.2.3 Experiments

	Predicted time	Test ping rtt	Test tcp rtt
loopback	0.1ms	0.110ms	79us
Remote machine	10ms	4.273ms	4.792ms

Table 4. Results

5.2.4 Analysis

From the measurement, we could know that most of our prediction is true except for the loopback tcp rtt. The reason for the time of ping is shorter than tcp rtt in remote machines is as I stated above. Because ping is based on ICMP, which is handled at kernel level, and could avoid the overhead of flow control or handshake etc. in TCP, so it is faster. Meanwhile, the user level program needs more context switches to use the tcp/ip stack in the kernel, which makes them much slower.
The reason why localhost tcp rtt is smaller than ping may be the fact that the operating system has optimized loopback packet sending. It is very likely that the OS realizes that the packet is sent to the localhost, so it is sent to the target process directly, which is slightly faster than using ping operation.

5.3 Peek bandwidth

5.3.1 Methodology

To measure the peak bandwidth, we could monitor file transmission and record the maximum bandwidth during the transmission. By simply setting a server and a client in the same or different machine, we could test the transmission bandwidth on a localhost or remote machine. To get the peak bandwidth, we test one transmision many times and use the maximum to represent the peak bandwidth.

5.3.2 Prediction

For loopback, the bandwidth has nothing to do with the network, and is limited by RAM bandwidth. So we estimate it with a speed in about 2GB/s. As for remote machines, the upper bound of bandwidth is limited by WIFI in LNA, the size of TCP packet is at most 65536Bytes, and the rtt we measured before is about 4.79ms, so we predict its peak bandwidth will be about 13.68MB/s.

5.3.3 Experiments

	Prediction	Test peek bandwidth
loopback	2GB/s	1429.55MB/s
Remote machine	13.68MB/s	9.53MB/s

Table 5. Results

5.3.4 Analysis

From our experiments, we could find that our loopback or remote peak bandwidth are both smaller than prediction, which could be caused by different reasons.

For loopback, because context switch overhead and tcp/ip stack overhead etc., we overestimate the performance of loopback peak bandwidth.

For remote machines, there can be many reasons. First of all, because TCP's 3-way and 4-way handshake, flow and congestion control etc, we couldn't keep transferring data at

peak bandwidth, sometimes there can be packet losses or congestion, and we only calculate the received valid date, rather than all of raw data. All of the above as well as context switch overhead shows that tcp/ip stack will produce considerable overhead that can not be ignored.

Another important reason is our WLAN environment. We use UCSD_PROTECTED WIFI in Geisel Library in UCSD, it is used by a lot of students at the same time, so our bandwidth can be limited and affected by other users in the same LNA, which would have an impact on our measurement on peak bandwidth.

5.4 Connection overhead

5.4.1 Methodology

In order to test connection overhead, we shall set two times before and after the action of setup or teardown. TCP establishes a connection by 3-way handshake, which is finished in the connect() system call in python, so just need to test the time difference before and after the connect action. Similarly, TCP terminates a connection by 4-way handshake, which is finished in close() system call, and we will do the same measure for the teardown part. For better accuracy, each measurement we will repeat 1000 times, and use the average as measured time.

5.4.2 Prediction

As we know, to establish a connection is much easier than terminating this connection. For the setup part, because 3-way handshake, so we estimate it will be 1.5rtt times for remote machines. As for loopback, we just predict the setup time the same as rtt, since there is no need for handshake in the same machine.

For the teardown part, because tcp could terminate the connection unidirectionally, it is much faster than establishing a connection. So we estimate teardown time as 0.1ms for both local and remote communication.

5.4.3 Experiments

	Measured time	Prediction
Remote setup	7.206ms	7.18ms
Remote teardown	78.2us	0.1ms
Localhost setup	61.38us	79us
Localhost teardown	11.46us	0.1ms

Table 6. Results

5.4.4 Analysis

For setup time, our estimation is quite accurate whether for loopback or remote machine. It makes sense because the connection action is based on tcp's 3-way handshake. The time for establishing connection is very close to 1.5RTT, and our experiment has proved it.

However, our teardown prediction is a little bit higher than actual measurements, especially for loopback teardown. We think it may because the action of close itself is asynchronous, the OS could terminate the connection very quickly by sending the FIN flag and leave the remaining jobs to the kernel. So our measurement is much less than prediction.

6.File system

6.1 Size of File Cache

6.1.1 Methodology

For an application accessing lots of file system data, an OS will use a notable fraction of main memories for the file system cache. The size of file cache is determined by the OS.

To show the effect of cache, we first need to read the entire file into memory. It will be stored in the cache area if the file size is no larger than the cache size. Then when next

time reading the file, data will be directly fetched from cache space instead of memory. To measure the boundary to cache size, we increased the file size granularly. The assumption is that after file size reaches the boundary, the file can no longer be stored in cache space. And the increment by cache effect will disappear. We expected to see I/O time per block approaching a number after some points.

6.1.2 Prediction

The machine we tested on has a total memory of 8 GB. OS can use a notable fraction of main memory as file system cache when it is needed. The OS itself will occupy some fraction of memory. By the time of testing, the activity monitor shows that window serve took about 500MB and the kernel task took about 300 MB. Among the memory used, 1.92 GB for weird memory and 1.55GB for Compressed. Though we had no idea about what these two memories are for, here we assume that cache size will not take these memories. So the total remaining memories available for file system cache is predicted to be no larger than 4 GB.

6.1.3 Results

We did experiments on file sizes of 100MB, 200MB, 500MB, 1GB, 2GB, 4GB, 5GB, 6GB, 6.5GB. We used *mkfile* in command line to create exact sizes of files. We used rdtsc() to get the starting and ending time of each operation. For each file size, we did 10 iterations and reported their average I/O read time per block.

We also took into consideration the effect of data prefetching, which will speed up the fetch operations by beginning a fetch operation whose result is expected to be needed soon. To avoid the effect of data prefetching, we read from the last block to the beginning.

Filesize	Time/μs
100MB	0.070002

200MB	0.04568
500MB	0.014158
1GB	0.006914
2GB	0.003439
4GB	0.001716
5GB	0.00138
6GB	0.001311
6.5GB	0.001276

Table 7. File Cache Size Results

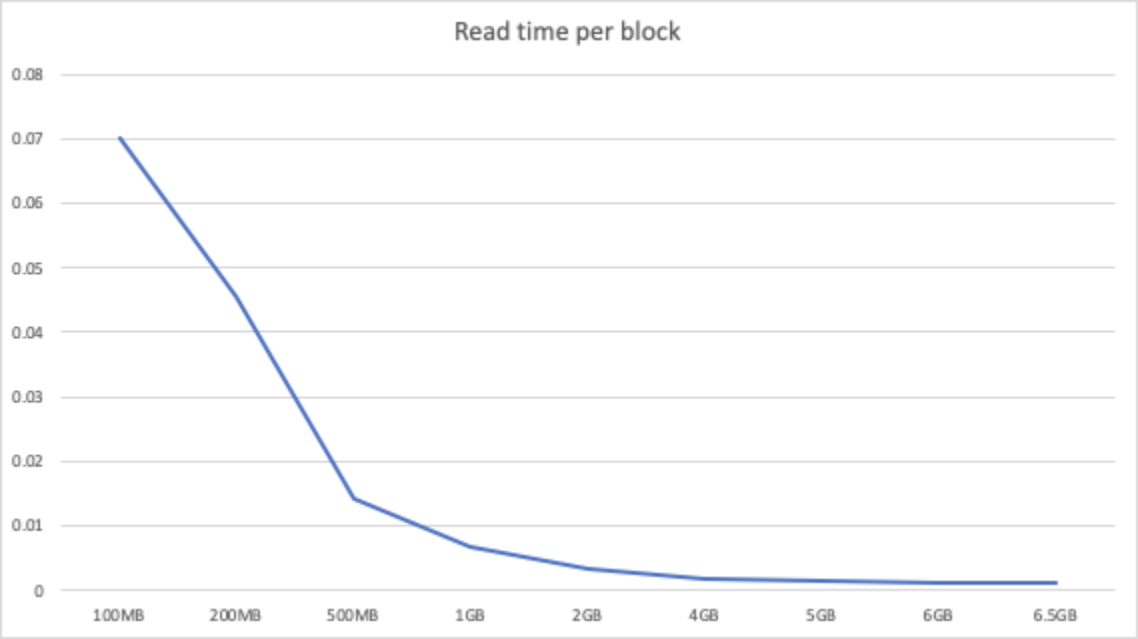

Figure 2. Cache Size Experiments

6.1.4 Analysis

As shown in the chart, there was a dramatic drop in the reading time for per block, which indicates that cache effect is keeping improving the I/O performance when file size increases. However, after the file size reached 4~5GB, the improvements slowed down. After 5GB, the read time per block becomes stable. This shows that, it had already reached the cache size boundary, and after that point, data was fetched from disk.

6.2 File read time

6.2.1 Methodology

We are going to compute file access time both by sequential and random, under different file sizes. In order to do that,we compute the access time based on a range of file sizes from 1MB - 128MB, doubled by order. For both sequential and random read, we set each read block size as 4*1024 and as usual, we use rdtsc() to compute each access time. Before reading, we should first get rid of the influence of file cache, using purge command to clean file cache. And then call fcntl(fd, F_NOCACHE, 1) before the actual read to make no use of file cache later on.

In sequential read, we read each file in order until the end. We repeat it for 100 times, calculating the average time. In random read, we also read the same size of file but instead of in order, we read from random offset. Similarly, we also repeat 100 times to get average.

6.2.2 Prediction

Since we conduct the experience on a machine with SSD, which has a much better performance than disk, we make the prediction based on the SSD reference. The major access time can be divided into two parts: target seeking time and data read time.

In sequential read, the seeking time may be negligible, so based on the SSD reference the read time is around 50us we predict.

In random read, there is seeking overhead by lseek(), the time is around 0.1ms, so in total, we predict that read should be around 150us.

6.2.3 Experiments

File size/MB	sequential/us	random/μs
1	62.7714	196.067
2	89.4813	232.958
4	71.0596	187.159
8	63.6504	189.515
16	66.7379	180.034
32	64.2089	188.656
64	63.4921	181.201
128	63.4974	178.233

Table 8. File read time Results

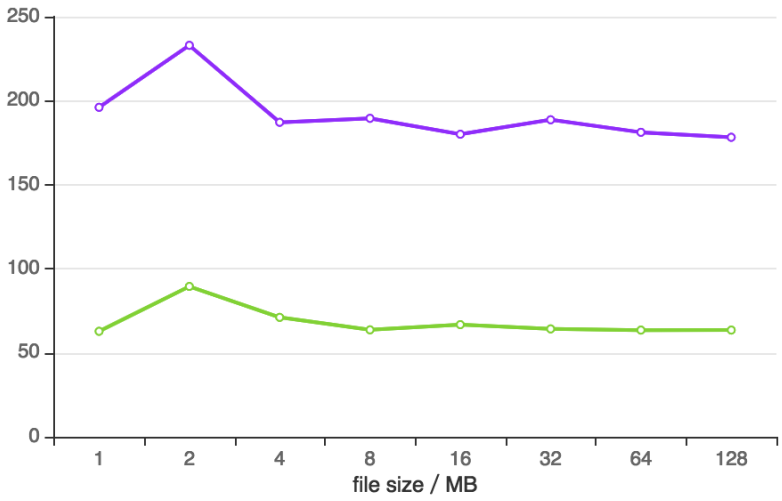

Figure 3. File Read Time Experiments

6.2.4 Analysis

According to the experiment data, we can see that the file size seems to have no influence on the read time. It is reasonable since that file cache is disabled, it is no different for file size on SSD. As we have predicted, random read has 100us more time than sequential read, due to the overhead of seeking (lseek).

6.3 Remote file read time

6.3.1 Methodology

In this part, we are going to evaluate the remote file read time, under the "network penalty". It is similar to the previous experiment, except that we should access the file on another machine. In order to do that, we use one machine as NFC server and mount the local directory to the remote server, then we can access the remote file by reading the local one. The connection is built on TCP and the operation we use to deal with cache fact should be also applied as the same in this experiment.

6.3.2 Prediction

In this case, the network facts we have measured above——Round trip time, Peak bandwidth, Connection overhead should all be taken into consideration. The actual data access time is almost the same with local access, which plays a much smaller role compared with the bottleneck, the network latency and bandwidth. On top of that, we predict the remote read time may be 6-7ms which is about 3 times of round trip time.

6.3.3 Experiments

File size/MB	sequential/ms	random/ms
1	6.2142	7.2310
2	6.3548	7.7261
4	6.5149	7.2092

8	6.3321	7.7351
16	6.5169	7.6827
32	6.9271	7.5839
64	5.8273	7.2480
128	6.4239	7.4973

Table 9. Remote File read time Results

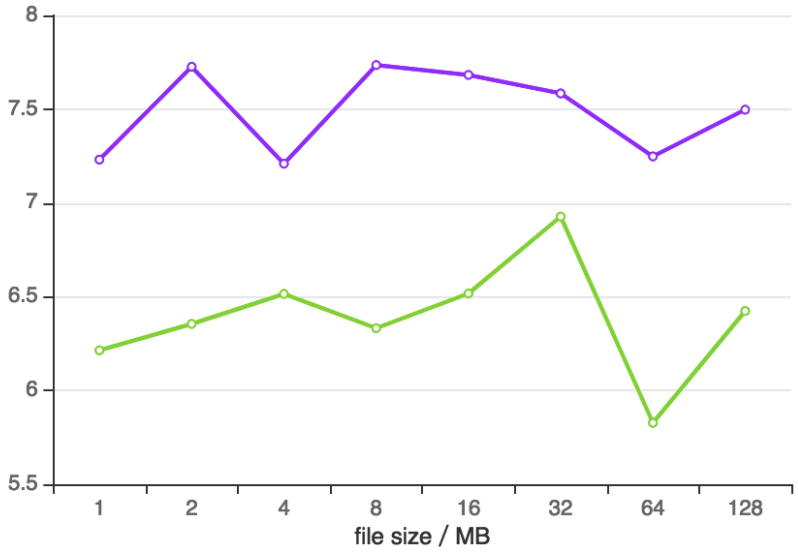

Figure 4. Remote File read time Experiments

6.3.4 Analysis

It is obvious that the file access time is negligible compared with network communication overhead. And we can see that the difference between sequential and random read is much smaller compared with total remote read time. So the remote file read time mostly depends on the network situation, which is usually unstable.

The network communication is costly: TCP will have to establish a reliable connection by a 3-Way Handshake Process before actually transferring the data. And then take the average transfer throughput (which we have evaluated in the former experiment) into consideration, the result of remote file read time is reasonable.

6.4 Contention

6.4.1 Methodology

To measure the file system contention, we created up to 10 processes. Let all processes simultaneously assess different files on the same disk. We kept all the processes running (reading from their file and printing each of their block read time). We then compared the average time to read one file system block of data as a function of the number of processes.

6.4.2 Prediction

As more processes perform the same operation on the same disk, the file contention will increase. However, we do not have an explicit idea about what the exact reading block time might be when there are N processes running. An general prediction is that, the per block reading time for each process is inversely proportional with the number of running processes. And we estimated that with 10 processes fetching data at the same time, the speed will slow down by 2~3 times as compared with 2 running processes.

6.4.3 Results

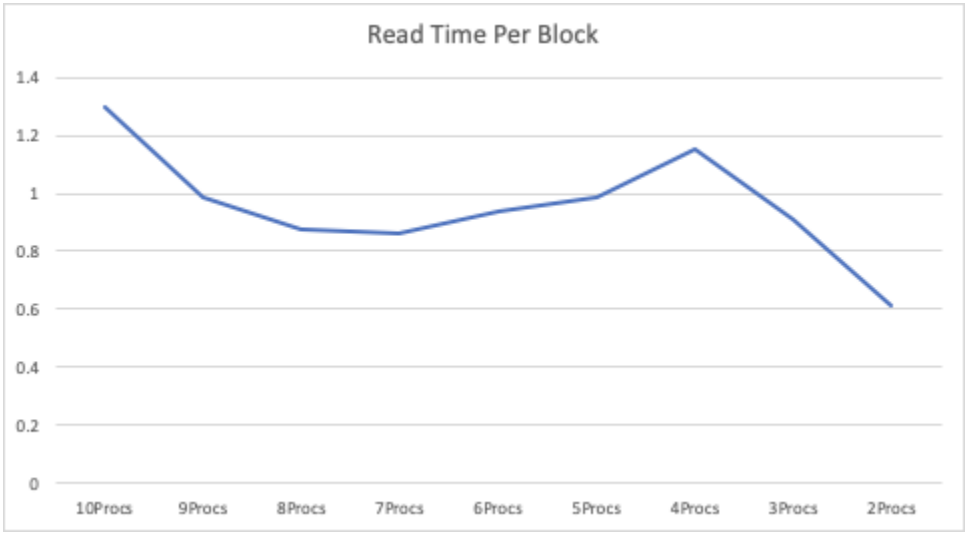

Figure 5. Contention Experiments

10 Procs	1.297031333
9 Procs	0.986540133
8 Procs	0.87765576
7 Procs	0.864444652
6 Procs	0.937640646
5 Procs	0.989618625
4 Procs	1.153134355
3 Procs	0.912170121
2 Procs	0.615996607

Table 10. Contention Results

6.4.4 Analysis

As the number of running processes increases, the average reading block time for each process generally increases, which indicates contention increases with more processes. However, there are some weird points such like 4 procs, which do not follow the general trend.

7 Reference

https://www.geeksforgeeks.org/socket-programming-python/

https://en.wikipedia.org/wiki/Internet_Control_Message_Protocol

https://en.wikipedia.org/wiki/Ping_(networking_utility)

https://en.wikipedia.org/wiki/Bandwidth_(computing)

https://en.wikipedia.org/wiki/TCP_window_scale_option

https://en.wikipedia.org/wiki/Transmission_Control_Protocol

https://stackoverflow.com/questions/27491314/how-to-measure-wallclock-time-of-file-i-o

https://www.geeksforgeeks.org/tcp-3-way-handshake-process/

https://www.wdiaz.org/how-to-mount-nfs-share-on-mac/

https://www.cyberciti.biz/faq/apple-mac-osx-nfs-mount-command-tutorial/

https://en.wikipedia.org/wiki/Memory_bandwidth

https://en.wikipedia.org/wiki/Mmap

https://pdfs.semanticscholar.org/cf87/3969cf371c8e9b9693b199bc582e1fdc1173.pdf

https://pdfs.semanticscholar.org/cf87/3969cf371c8e9b9693b199bc582e1fdc1173.pdf

https://www.geeksforgeeks.org/page-fault-handling-in-operating-system/

https://en.wikipedia.org/wiki/Page_fault

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

https://course.ece.cmu.edu/~ece740/f11/lib/exe/fetch.php?media=wiki:lectures:onur-740-fall11-lecture24-prefetching-afterlecture.pdf