System Measurement

Sebastian Cheah
Olga Souverneva
Nico Pappagianis

Introduction

Our goal is to analyze the hardware components of our base machine, and determine the effects they have on operating system services for the Windows 8.1 Operating System. We will create and perform experiments in order to further understand operating system services. These experiments will be implemented and measured in C++. Our compiler is Microsoft Visual C++ (cl.exe) product version (12.00.31101.0).

The team completed the project in approximately 90 hours with the time being some-what weighted towards the later operations of the project. Our estimate per subsystem is shown in the table below.

Subsystem	Approximate time spent (hrs)
CPU, Scheduling, and OS Services	18
Memory	24
Network	24
File System	24

The teams division of labor for this project is listed in the table below.

Subsystem	Operations	Responsible Team Member(s)
CPU, Scheduling, and OS Services	Measurement overhead	Sebastian Cheah
	Procedure call overhead	Sebastian Cheah
	System call overhead	Sebastian Cheah, Olga Souverneva
	Task creation time	Sebastian Cheah, Olga Souverneva
	Context switch time	Sebastian Cheah
Memory	RAM access time	Sebastian Cheah
	RAM bandwidth	Olga Souverneva
	Page fault service time	Olga Souverneva
Network	Round trip time	Sebastian Cheah
	Peak bandwidth	Sebastian Cheah
	Connection overhead	Sebastian Cheah
File System	Size of file cache	Olga Souverneva
	File read time	Olga Souverneva
	Remote file read time	Olga Souverneva
	Contention	Olga Souverneva

Machine Description

1. Processor:
    - Model: Intel Core i5 4300U @ 2.49GHz (maximum)
    - Cycle time: 0.4ns
    - L1 cache: 128 KB 8-way; Data: 2x32 KB; Instruction: 2x32 KB
    - L2 cache: 2x256 KB 8-way
    - L3 cache 3 MB 12-way
2. Memory bus
    - 800 MHz
3. RAM size
    - 8 GB
4. Disk: capacity, RPM, controller cache size
    - 240 GB SSD
    - No on disk cache
5. Network card speed
    - 450 Mbps
6. Operating system (including version/release)
    - Name: Microsoft Windows 8.1
    - Version: 6.3.9600

Experiment Setup

We disable dynamically adjusted CPU frequency and set it to the maximum so that the frequency at which the processor computes is deterministic and does not vary. We set the process priority as "High" for our benchmarks. We also restrict our measurement programs to using a single core.

Example use:

```
start /high /affinity 1 program.exe
```

Visual Studio also provides release build options which includes compiler optimizations such as /O2.

CPU, Scheduling, and OS Services

Measurement overhead

<u>Predictions:</u>

Since we will be using the Windows API, we estimate the hardware overhead for measuring time to be approximately 12 cycles (4.8 ns) to account for a procedure call, control transfer, and moving the value to a 64-bit register. We estimate the software overhead to be an additional 20 cycles (8 ns) to account for validating the input and output parameters as well as additional processing of the variables.

For loop overheads, we estimate hardware overhead to be 5 cycles (2 ns) as zero overhead looping is not a feature of this processor. Some overhead has to be taken up by performing the counter increment and branching to the loop start address. We estimate additional software overhead to be 20 cycles (8 ns).

<u>Method:</u>

To measure time for our experiments we intend to use the Windows API advocated by Windows: *QueryPerformanceCounter* [1]. While this is an API call and slower than RDTSC, it is stated that it achieves a measurement resolution of <1 us and may be easier to use than RDTSC due to its frequent rollover [2]. We will make two calls to *QueryPerformanceCounter* and subtract the difference if the API succeeds. We will then convert the cycle count to time estimates using the CPU frequency. To obtain an accurate cycle time measurement we plan to enclose this operation in a loop and use a large number of

The overhead of measurements taken in a loop will be obtained by enclosing a simple for loop in starting and ending timestamps.

Psuedocode:

```
double measurement_overhead() {
        __int64 iTotalTime = 0;
        for (int i = 0; i < NUM_TESTS; i++) {
                QueryPerformanceFrequency(&freq);
                QueryPerformanceCounter(&start_time_stamp);
                // do nothing
                QueryPerformanceCounter(&end_time_stamp);
                iTotalTime += time_elapsed();
        }
        double dTotalTime = static_cast<double>(iTotalTime);
        return dTotalTime / NUM_TESTS;
}

double loop_overhead() {
        __int64 iTotalTime = 0;
        for (int i = 0; i < NUM_TESTS; i++) {
                QueryPerformanceFrequency(&freq);
                QueryPerformanceCounter(&start_time_stamp);
                for (int j = 0; j <= 0; j++) {}
                QueryPerformanceCounter(&end_time_stamp);
                iTotalTime += time_elapsed();
        }
        double dTotalTime = static_cast<double>(iTotalTime);
        return dTotalTime / NUM_TESTS – measurement_overhead();
}
```

Results:

Overhead	**Measured Time (ns)**	**Estimated Time (ns)**
Measurement	15.826	12.8
Loop*	1.148	10

*Measurement overhead subtracted from result

Discussion:

Our prediction for the measurement overhead was fairly accurate after breaking it down into instructions involving a procedure call, control transfers, and reads and writes to 64-bit registers. We were accurate in a sense that loop overheads are lower than measurement overheads. However, we overestimated the overheads associated with a loop. Additionally, we did not consider branch prediction since our loop test construct only loops once. If it was run more than once, the measured time would be even lower than the one reported above due to our linear counter incrementing. There may be less instructions involved in our loop due to a fixed starting address and linear counter incrementing.

Procedure call overhead

Predictions:

The overhead for the procedure call is largely hardware driven as the procedure call would cause a jump to a procedure defined elsewhere and thus the majority of the overhead lies in processor instructions. We estimated that the hardware overhead of a procedure call should be around 5 CPU cycles (2 ns) to account for storing the data in registers and the jump. We estimate software overhead to be an additional 20 cycles (8 ns) to process the procedure call and its arguments.

We predict the individual hardware overheads and software overheads to increase linearly (5 cycles = 2 ns) with the number of inputs, incurring a total incremental overhead of 4 ns.

Method:

To measure the procedure call overhead we will record a time stamp. Then immediately call a procedure with the specified input count. The procedure will return without processing the input parameters and we will record the end time stamp.

Psuedocode:

```
double procedureTime(int numArgs) {
        __int64 iTotalTime = 0;
        for (int i = 0; i < NUM_TESTS; i++) {
                switch (numArgs) {
                case 0:
                        QueryPerformanceFrequency(&freq);
                        QueryPerformanceCounter(&start_time_stamp);
                        procedure0();
                        QueryPerformanceCounter(&end_time_stamp);
                        break;
                case 1:
                        QueryPerformanceFrequency(&freq);
                        QueryPerformanceCounter(&start_time_stamp);
                        procedure1(1);
                        QueryPerformanceCounter(&end_time_stamp);
                        break;
                case 2:
                        QueryPerformanceFrequency(&freq);
                        QueryPerformanceCounter(&start_time_stamp);
                        procedure2(1, 2);
                        QueryPerformanceCounter(&end_time_stamp);
                        break;
                case 3:
                        QueryPerformanceFrequency(&freq);
                        QueryPerformanceCounter(&start_time_stamp);
                        procedure3(3, 2, 1);
                        QueryPerformanceCounter(&end_time_stamp);
                        break;
                case 4:
                        QueryPerformanceFrequency(&freq);
                        QueryPerformanceCounter(&start_time_stamp);
                        procedure4(1, 2, 3, 4);
                QueryPerformanceCounter(&end_time_stamp);
                break;
            case 5:
                QueryPerformanceFrequency(&freq);
                QueryPerformanceCounter(&start_time_stamp);
                procedure5(5, 4, 3, 2, 1);
                QueryPerformanceCounter(&end_time_stamp);
                break;
            case 6:
                QueryPerformanceFrequency(&freq);
                QueryPerformanceCounter(&start_time_stamp);
                procedure6(1, 2, 3, 4, 5, 6);
                QueryPerformanceCounter(&end_time_stamp);
                break;
            case 7:
                QueryPerformanceFrequency(&freq);
                QueryPerformanceCounter(&start_time_stamp);
                procedure7(7, 6, 5, 4, 3, 2, 1);
                QueryPerformanceCounter(&end_time_stamp);
                break;
            default:
                return 0; //not valid
                break;
            }
            iTotalTime += time_elapsed();
        }
        double dTotalTime = static_cast<double>(iTotalTime);
        return (dTotalTime / NUM_TESTS) - measurement_overhead();
}
```

Results:

# Arguments	Measured Time (ns)	Estimated Time (ns)	Measured Incremental overhead (ns)	Estimated Incremental Overhead (ns)
0	10.004	10	0	0
1	10.578	14	0.574	4
2	17.794	18	7.216	4
3	18.532	22	0.738	4
4	12.2592	26	-6.2728	4
5	19.885	30	7.6258	4
6	21.32	34	1.435	4
7	21.73	38	0.41	4

Note: Measurement overheads were subtracted from measured time

Discussion:

We are accurate in the assumption that with more arguments, there would be an increase in time taken to process the procedure call. However, these results are variable, and an anomaly appeared when measuring a procedure call with 4 arguments. Interestingly enough, the incremental overhead does spike above and below our predicted numbers. After equalizing these spikes, it seems the incremental overhead does hover around 2 cycles (0.8 ns).

System call overhead

Predictions:

The software overhead with system calls is greater than that of procedure calls as the caller must wait for the operating system to perform the requested activity. In the case of opening the file, the operating system must check that the caller has permission to access the file. We estimate the software overhead to be on the order of 500 CPU cycles (200 ns).

There is also some hardware overhead as during a system call the hardware raises the privilege level of the CPU and transfers control for the kernel mode transition. On the Intel processor this is done with the SYSENTER/SYSEXIT fast entry instructions. This avoids interrupt latency, but we still estimate a hardware overhead of 20 cycles (8 ns).

Method:

We considered using *GetCurrentThreadId* or *GetCurrentProcessId* initially, but the measurement results when using these functions were much lower than expected. The low results may suggest that there is no transition from user to kernel mode, implying that *GetCurrentThreadId* and *GetCurrentProcessId* are not system calls, and may only have to fetch data within user-space.

To measure the overhead of a system call we will use the Windows *GetThreadId* API for an arbitrary thread. We will obtain time stamps before and after the API call. We selected this API as it involves a kernel mode transition with minimal subsequent steps. To avoid caching interfering with our measurements we plan to use a new thread each time.

Psuedocode:

```
double system_overhead()
{
       __int64 iTotalTime = 0;
       // Create the thread
       DWORD tid = 0;
       HANDLE hid = CreateThread(
              NULL,                    // default security attributes
              0,                       // use default stack size
              MyThreadFunction,        // thread function name
              NULL,                    // argument to thread function
              CREATE_SUSPENDED,        // create, BUT do not run!
              &tid);                   //
       for (int i = 0; i < NUM_TESTS; i++) {
              // Start timing
              QueryPerformanceFrequency(&freq);
              QueryPerformanceCounter(&start_time_stamp);
              GetThreadId(hid); // system call
              QueryPerformanceCounter(&end_time_stamp);
              iTotalTime += time_elapsed();
       }
       double dTotalTime = static_cast<double>(iTotalTime);
       return (dTotalTime / NUM_TESTS) - measurement_overhead();
}
```

Results:

Overhead	Measured Time (ns)	Estimated Time (ns)
System Call*	330.065	208

*Measurement overhead subtracted from result

Discussion:

Our estimated time turned out to be lower than the actual result. We may have underestimated the overhead of transitioning between user and kernel mode. Additionally, it may be possible that *GetCurrentThreadId* is not as minimal as a system call as we have hoped.

Task Creation time

Predictions:

For creating a new process, the OS has to provide a self-contained environment to execute in. As such, each process has its own memory space and resources, which has to be copied/created when we perform an experiment where a parent process creates another process.

For creating a new thread, the thread will exist within the parent process, and side by side with other threads. As such, the new thread will share the process's memory and resources with other threads and vice versa. As such, creating a new thread would require less resources than creating a new process.

We estimate that creating a new thread will take on the order of 1000 CPU cycles (400 ns).

We estimate that creating a new process will take three orders of magnitude more, on the order of 1,000,000 CPU cycles (400000 ns).

Method:

Our experiment will make key use of the windows API functions *CreateProcess* and *CreateThread*. We will take the starting timestamp before the creation call and take the ending timestamp after creation. Note that the thread/process created will not be executed, but rather initialized in a suspended state.

Psuedocode:

```
double threadTest()
{
        __int64 iTotalTime = 0;
        for (int i = 0; i < NUM_TESTS; i++) {
                // Start timing
                QueryPerformanceFrequency(&freq);
                QueryPerformanceCounter(&start_time_stamp);
                // Create the thread
                DWORD tid = 0;
                HANDLE hid = CreateThread(
                        NULL,                   // default security attributes
                        0,                      // use default stack size
                        MyThreadFunction,       // thread function name
                        NULL,                   // argument to thread function
                        CREATE_SUSPENDED,       // create, BUT do not run!
                        &tid);                  // returns the thread identifier
                QueryPerformanceCounter(&end_time_stamp);
```

```
			CloseHandle(hid);
			iTotalTime += time_elapsed();
		}
		double dTotalTime = static_cast<double>(iTotalTime);
		return (dTotalTime / NUM_TESTS) - measurement_overhead();
}

double processTest()
{
		STARTUPINFO si;
		PROCESS_INFORMATION pi;
		__int64 iTotalTime = 0;
		for (int i = 0; i < NUM_TESTS; i++) {
			ZeroMemory(&si, sizeof(si));
			si.cb = sizeof(si);
			ZeroMemory(&pi, sizeof(pi));
			// Start timing
			QueryPerformanceFrequency(&freq);
			QueryPerformanceCounter(&start_time_stamp);
			// Create process and thread
			if (!CreateProcess(L"C:\\Windows\\notepad.exe",
				NULL,
				NULL,
				NULL,
				FALSE,
				CREATE_SUSPENDED,
				NULL,
				NULL,
				&si,
				&pi)
				)
			{
				return -1;
			}
			// Get the end time
			QueryPerformanceCounter(&end_time_stamp);
			// Terminate process and thread
			TerminateProcess(pi.hProcess, NULL);
			CloseHandle(pi.hThread);
			iTotalTime += time_elapsed();
		}
		double dTotalTime = static_cast<double>(iTotalTime);
		return (dTotalTime / NUM_TESTS) - measurement_overhead();
}
```

Results:

Task Creation	Measured Time (ns)	Estimated Time (ns)
Thread	14233.986	400
Process	434520.572	400000

Discussion:

Our estimated time for process creation was very close, but we based it upon its comparison to thread creation. We greatly underestimated thread creation, which seems to involve more complexity than predicted.

On the other hand, we were correct in our logic when we predicted process creation to take more time than thread creation.

Context switch time

Predictions:

Context switching varies between processes and threads. In process would incur costs involving the TLB. This is because a context switch between two different memory spaces would invalidate virtual address to physical address translation mappings. Flushing the TLB cache and rebuilding incurs a cost, influencing our results. For threads, they share the same memory space, meaning the translation mappings will remain valid and therefore no flushing of the TLB is needed.

In terms of hardware overhead, we expect a difference between thread and process context switching to be around a factor of 100. For base hardware overhead, we estimate 500 cycles (200 ns) for thread switching and 500000 cycles (200000 ns) for process switching. We expect software overheads between threads and software to have a smaller factor. We expect the base software overhead to be more than hardware for threads, but less for threads. We estimate 1000 cycles (400 ns) for thread scheduling and 2000 cycles (800 ns) for process scheduling.

Method:

Our experiment will make key use of the windows API functions *ResumeThread*. This will cause the arbitrarily created thread or process on the current process/thread to begin execution on another thread or process that is ready to run. To better measure our context switch time we also use *SetThreadPriority* and *SetPriorityClass* in order to lessen scheduler overheads. We will take the starting timestamp before the switch call, and the process/thread's task will be to execute, where we will take the ending timestamp for our measurements. For thread communication, it is easy to communicate time stamps. For processes, we had to use named pipes in order to implement inter-process communication. We carefully choose when to write our time stamps in order to avoid additional communication overheads.

Psuedocode:

Thread context switch:

```
double thread_context_switch()
{
        __int64 iTotalTime = 0;
        for (int i = 0; i < NUM_TESTS; i++) {
                DWORD tid = 0;
                HANDLE hid = CreateThread(
                        NULL,                     // default security attributes
                        0,                        // use default stack size
                        MyThreadFunction,         // thread function name
                        NULL,                       // argument to thread function
                        CREATE_SUSPENDED,         // create, BUT do not run!
                        &tid);                              // returns the thread identifier
```

```
            SetThreadPriority(hid, 10); // set high priority for scheduler
            // Start timing
            QueryPerformanceFrequency(&freq);
            QueryPerformanceCounter(&start_time_stamp);
            ResumeThread(hid);
            WaitForSingleObject(hid, INFINITE);
            iTotalTime += time_elapsed();
        }
        double dTotalTime = static_cast<double>(iTotalTime);
        return (dTotalTime / NUM_TESTS) - measurement_overhead();
}
DWORD WINAPI MyThreadFunction(LPVOID lpParam)
{
        QueryPerformanceCounter(&end_time_stamp);
        return 0;
}
```

Process context switch:

```
double process_context_switch()
{
        STARTUPINFO si;
        PROCESS_INFORMATION pi;
        __int64 iTotalTime = 0;
        for (int i = 0; i < NUM_TESTS; i++) {
            ZeroMemory(&si, sizeof(si));
            si.cb = sizeof(si);
            ZeroMemory(&pi, sizeof(pi));
            // Start timing
            // Create process and thread
            LPTSTR szCmdline = _tcsdup(TEXT("CPU_Dummy_Child_Process.exe"));
            if (!CreateProcess(NULL,
                    szCmdline,
                    NULL,
                    NULL,
                    FALSE,
                    CREATE_SUSPENDED,
                    NULL,
                    NULL,
                    &si,
                    &pi)
                    )
            {
                    return -1;
            }

            QueryPerformanceFrequency(&freq);
            QueryPerformanceCounter(&start_time_stamp);
            ResumeThread(pi.hThread);
            WaitForSingleObject(pi.hThread, 2000);
            LPTSTR lpszPipename = TEXT("\\\\.\\pipe\\mypipe");
            // Try to open a named pipe; wait for it, if necessary.
            HANDLE hPipe;
            while (1)
            {
                    hPipe = CreateFile(
                            lpszPipename,   // pipe name
                            GENERIC_READ,
                            FILE_SHARE_READ,
                            NULL,           // default security attributes
                            OPEN_EXISTING,  // opens existing pipe
                            0,              // default attributes
                            NULL);          // no template file

                    // Break if the pipe handle is valid.
                    if (hPipe != INVALID_HANDLE_VALUE)
                            break;
            }
            char buffer[21];
            DWORD read = 0;
            while (1) {
                    if (!ReadFile(
                            hPipe,
                            buffer,           // read data
                            sizeof(buffer) - 1, // max length (leave room for terminator)
                            &read,            // bytes actually read
                            NULL))
                            continue;
                    else
                            break; //success!
            }
            __int64 procEnd = _atoi64(buffer);
            iTotalTime += time_elapsed(start_time_stamp.QuadPart, procEnd);
            TerminateProcess(pi.hProcess, NULL);
            CloseHandle(pi.hThread);
            CloseHandle(hPipe);
        }
        double dTotalTime = static_cast<double>(iTotalTime);
        return (dTotalTime / NUM_TESTS) - measurement_overhead();
}

int dummychild_tmain(int argc, _TCHAR* argv[])
{
        QueryPerformanceFrequency(&freq);
        QueryPerformanceCounter(&end_time_stamp);
        HANDLE pipe = CreateNamedPipe(
            L"\\\\.\\pipe\\mypipe",                  // name of the pipe
            PIPE_ACCESS_OUTBOUND | FILE_FLAG_WRITE_THROUGH,                //
            PIPE_TYPE_MESSAGE | PIPE_WAIT, PIPE_UNLIMITED_INSTANCES,
            512, 512, 1000, NULL);                                  // default
        if (pipe == INVALID_HANDLE_VALUE) return -1; //error
        char line[21];
        _i64toa_s(end_time_stamp.QuadPart, line, 21, 10);
        DWORD written = 0;
        int test = strlen(line);
        while (1) {
            if (WriteFile(
                    pipe,
                    line,         // sent data
                    strlen(line), // data length
                    &written,     // bytes actually written
                    NULL))
                    break;

        }
```

```
        DisconnectNamedPipe(pipe);
        CloseHandle(pipe);
        return 0;
}
```

Results:

Context Switch	Measured Time (ns)	Estimated Time (ns)
Thread	26192.56	600
Process	2879458.56	200800

Discussion:

Our estimates were off when predicting base costs. On the other hand, it seems the relationship between thread and process context switching times have a difference by a factor of ~109. The exact disparity between hardware and software overheads cannot be seen by these results alone. We do not expect software overheads between threads and processes to have the same factor as hardware overhead. Therefore, we believe the hardware overheads to differ by a wider margin than expected.

Memory operations

RAM access time

Predictions:

We will predict RAM access times with the following hardware: L1 cache (128 KB), L2 cache (512 KB), L3 cache (3 MB) and main memory (8 GB). We expect timings to change when accesses start to go down the memory access hierarchy. The cache accesses are dominated by the distance between CPU and memory module, which we will try to account for in the estimations. For an L1 cache access, we estimate the measurements to be fast, around 10 cycles (4 ns). For an L2 cache access compared to an L1 cache access, we estimate it would increase non-linearly due to the distance between CPU and memory and predict 30 cycles (12 ns). For an L3 cache access, we predict with the same scheme and estimate 50 cycles (20 ns). For main memory, there will be significant distance leading to our estimate of 200 cycles (80 ns).

It is said that there is no software overhead when measuring RAM access latencies. This is because a measured time is pure latency time and may be zero despite the load instruction executing in non-zero time. One report says, “Zero is defined as one clock cycle; in other words, the time reported is only memory latency time, as it does not include the instruction execution time.” [3]. Therefore, we do not predict software overhead will influence our measurements.

Method:

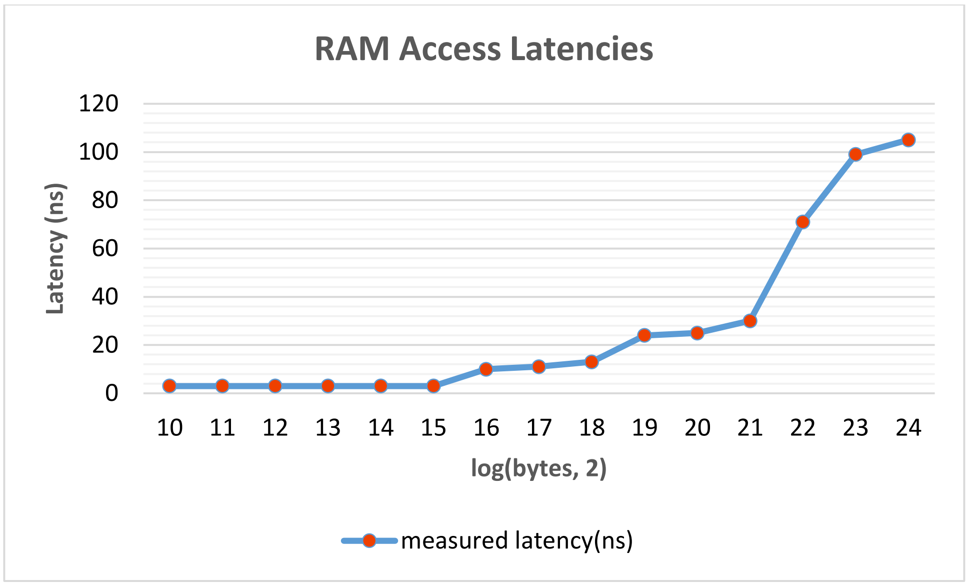
Initially, we first tested by accessing an array filled with integer accesses to an array of integers. Initial tests with sequential accessing reveal a near equal amount of time per access regardless of array size. We noticed this was the result of the cache prefetcher, and then opted to implement a stride, which specifies the amount of skipped memory before the next access. To further implement this, we create a linked list, making the compiler think that the next element to fetch is dependent on the data of the current element. After implementing this, there was another problem that arose when applying compiler optimizations (/O2) to the code. The compiler notices that the code actually does nothing when accessing memory, resulting in invalid results after optimization. To rectify this, we implement a mechanism (independent of the timing) similar to Lmbench's use_pointer() function, which makes the compiler think the code needs to be run and restores our code's functionality [3]. We perform one million accesses, and average out the access times.

Psuedocode:

```
int *arr = (int*)malloc(sizeof(int) * num_elements);
//set up data path to avoid prefetching, stride-based access
//Key is linked list with nodes allocated stride bytes apart in a contiguous memory space
pointer = setup_circular_linked_list();
start_time_stamp(&start);
for (i = 0; i < num_accesses/1000; i++) { // minus overhead when using time_elapsed()
       THOUSAND // macro for 1000 unrolled accesses to circular linked list
}
end_time_stamp(&end)
use_pointer((void*)p); //to prevent compiler thinking this is a dead pointer
free(arr);
return time_elapsed() / num_accesses;
```

Results:

array size(bytes)	measured latency(ns)	Estimated latency(ns)
1024	3	4
2048	3	4
4096	3	4
8192	3	4
16384	3	4
32768	3	4
65536	10	4
131072	11	4
262144	13	12
524288	24	12
1048576	25	20
2097152	30	20
4194304	71	80
8388608	99	80
16777216	105	80



L1 Cache is at data point 2^17 bytes (128 KB)

L2 Cache is at data point 2^19 bytes (512 KB)

L3 Cache is between data points 2^21 - 2^22 bytes (3 MB)

Discussion:

When designing our experiment, we were concerned with the cache prefetching policies. Documentation on policies are sparse, but it seems that these results reveal that our hardware does not possess a stride sensitive prefetcher. These results are successful in the fact that they show noticeable changes in latency, which signify the size of our caches. The results do spike between cache sizes, but it seems there is some memory being used by other processes other than our benchmark, causing measurements to jump at lower than expected array sizes.

RAM bandwidth

Predictions:

Hardware specifications indicated that the bandwidth was 12800 MB/S by performing the following calculation:

Hardware Bandwidth = Base Clock * Bus Width * Data Rate * 1 Byte / 8 Bits
Hardware Bandwidth = 800 MHz * 64 bits * 2 * 1 Byte/ *Bits
Hardware Bandwidth = 12,800 MB/S

We estimate and additional 20% software overhead may occur. We estimated that both reading and writing would have similar software overheads.

Method:

To measure the RAM bandwidth we looked to perform a large number of memory accesses where each access was likely to ensure a cache miss. Total cache size for the processor was 3.832 MB. Thus we created an array that was the ceiling of 2 times the total cache (8 MB). We accessed one value at a random index in the array, and another value at least half the array size away in memory to ensure that cache misses occurred. To test reading, we added a dummy operation to add the first and second read values. To test writing, we wrote the index of the first value to the second and vice versa. We used macros for loop unraveling to avoid loop overhead.

We measured the time necessary to read and write 50 thousand 32-bit integer values representing 200 kB of data. We also measured our measurement overhead due to the random number generation, addition, and procedure calls by repeating the same code but without reading or writing to the array. We subtracted the overhead from the measured time to obtain the corrected read time. The reported bandwidth is the bytes read or written divided by the corrected time.

Pseudocode:

```
big array is an 8MB array of integers initialized to arbitrary values.
data size is the count of integers in big array.
int ReadTime {
    Take start time stamp
    *add big array[rand()%data size] and big array[((rand()%data size)+data size/2)%data size]
    repeat 25000 times from *
    Take end time stamp
    output elapsed time.
}
int ReadTimeOverhead {
    Take start time stamp
    *Add (rand()%data size) and (((rand()%data size)+data size/2)%data size)
    Repeat 25000 times from *
    Take end time stamp
    output elapsed time.
}
int WriteTime {
    int index1, index2
    Take start time stamp
    *index1<- rand()%data size
    index2<-((index1)+data size/2)%data size]
    big array[index1] = index2
    big array[index2] = index1
    repeat 25000 times from *
    Take end time stamp
    output elapsed time.
}
int WriteTimeOverhead {
    int index1, index2
    Take start time stamp
    *index1<- rand()%data size
    index2<-((index1)+data size/2)%data size]
```

```
        Repeat 25000 times from *
        Take end time stamp
        output elapsed time.
}

report read bandwidth as 200kB/(ReadTime- ReadOverheadTime)
report writebandwidth as 200kB/(WriteTime- WriteOverheadTime)
```

Results:

The averaged results of 100 runs are reported in the table below.

	Hardware Specification (MB/s)	Softaware Overhead Estimate (%)	Estimated Bandwidth (MB/s)	Measured Bandwidth (MB/s)
Read	12800	20%	10240	389
Write	12800	20%	10240	628

Discussion:

Given that we are observing nearly double the bandwidth for memory writes versus reads we feel that it may be the case that memory reads are experiencing an oddly high number of cache misses, which impair the read bandwidth severely. This may be due to our methodology used in testing the RAM bandwidth such that our system is incurring an imbalanced cost for reads versus writes.

Page fault service time

Predictions:

Hardware specifications for a solid state drive (SSD) indicate that it takes approximately 25 µs to fetch a page from the IO on a read. We expected software overhead to contribute minimally to this amount. Because the SSD on our test system does not contain an on-disk cache we expect that our measurements might be somewhat high.

Method:

To measure the page fault time, we reserved a large region spanning 100 pages of memory for a dynamic array using the windows VirtualAlloc API [4]. We then iterated through the array and used dynamic exception handling to catch when a page fault exception occurred. We then used the VirtualAlloc API to page in the next page. This included the very first write attempt. We measured the time it took for the API to return using QueryPerformanceCounter. We then reported the averaged time.

Pseudocode:

```
        Reserve PAGE_COUNT pages in the virtual address space using VirtualAlloc.
        int page fault count
        page fault count <- 0
        int page fault time
        page fault time <- 0
```

```
For  (i= start address of the reserved space; i<end of reserved address space; i++)
    Try writing a value to the ith address.
    If a page fault occurs:
        Catch exception.
        Start timing .
        Fetch next page using VirtualAlloc.
        End timing.
        fault count++
        fault time += elapsed time
    End if
End for
Output (fault time/ fault count)
```

Results:

	Predicted Overhead (µS)	Measured Overhead (µS)
Page Fault Overhead	25	12.89

Discussion:

One possible explanation for our low page fault overhead is that the predictions we made were based on measurements made with slightly older SSDs or perhaps a lower bandwidth interface. Due to the lack of an on-disk cache we are curious to investigate how this test runs on a similar SSD with an on-disk cache, although none were available at the time of testing.

Network

For the below tests, we implement a separate client programs for use in interacting with a flexible TCP server that can either receive messages only, or act as an echo server. Our version of the TCP server is used in every network test. When running remote communication tests, the TCP server is run on an identical machine on the same computer network to reduce extraneous variables affecting results.

Round Trip Time

Predictions:

For our predictions, we will measure in terms of milliseconds (see Method section).

We predict TCP round trip times to be higher than ICMP. This is in part due to the network flow when using TCP protocol. ACK packets are used to acknowledge receipt of data packets while in a TCP session. There is also a difference in the extra amounts of data needed by the network protocol, which is stored and send within each packet. The minimum size of TCP headers is 20 bytes [5]. On the other hand, ICMP headers for echo requests and replies are 8 bytes [6]. These packet sizes will affect the network card's ability to write/read data on the wire.

In terms of software overhead, the API calls for TCP (send) and ICMP (ICMPSendEcho) are synchronous functions. In TCP, since ACK packets are used in the reply from the server after a send function, an additional function (recv) has to be called to receive the actual reply message. With this in mind, we expect the software overhead for TCP to be at least double that of ICMP due to the blocking nature of a send/recv pair.

We must also consider comparing loopback and remote connections using TCP and ICMP. With remote connections, we send data at a theoretical maximum of 450 mbps. On the other hand, a loopback connection would involve buffering data within the kernel without reading/writing on the network wire.

With all these considerations in mind, for remote connections, we predict a software overhead of 1 ms for ICMP, and 2 ms for TCP. In terms of hardware overhead, we predict a relation between TCP and ICMP, with 0.5 ms for ICMP and 1 ms for TCP. For loopback connections, we predict zero hardware overheads, and the same amount of software overheads as remote connections.

Method:

For this experiment, we set our TCP server to echo back whatever it receives, simulating the behavior of an ICMP echo request.

Two different network protocols are used: TCP and ICMP. We use Winsock to access network services and utilize both protocols [7]. For TCP, we must consider the window size. The window size specifies the maximum number of bytes sent within the round trip time. A TCP packet header allows a maximum window size of 65535 bytes. As such, our experiment involves sending 4 KB of data between client and server in both protocols for consistency and avoiding fragmented packets. The round trip time is manually measured beginning when a TCP client initiates a send, and ending after receiving all of the intended data. This way, we avoid the connection overhead in our measurements (see Connection Overhead). For ICMP, round trip time is actually measured automatically when using said protocol. However, the ICMP protocol only measures in milliseconds, so we do the same for our TCP measurements in order to keep the precision consistent between the two measurements.

We sent 100 packets in our tests, and took the average round trip time of our results.

Pseudocode:

```
double tcp_rtt() {
        sock = setupConnection()
        __int64 iTotalTimeElapsed = 0;
        for (int i = 0; i < NUM_PACKETS; i++)  {// For each packet
                QueryPerformanceFrequency(&freq);
                QueryPerformanceCounter(&start_time_stamp);
                int iSent = send(sock, szSendMessage, strlen(szSendMessage), 0);
                int iRecv = recv(sock, szBuffer, iBytesToRecv, 0);
                QueryPerformanceCounter(&end_time_stamp);
                iTotalTimeElapsed += time_elapsed();
        }
        double dtotalTime = static_cast<double>(iTotalTimeElapsed);
        return dTotalTime/NUM_PACKETS;
}

double icmp_rtt() {
        __int64 totalRTT = 0;
        for (int i = 0; i < NUM_PACKETS; i++) {
                dwRetVal = IcmpSendEcho(hIcmpFile, ipaddr, szSendMessage,
                sizeof(szSendMessage), NULL, lpReplyBuffer, dwReplySize, 5000); // 5s
                timeout
                if (dwRetVal == 0) { //error
                       return -1;
                }
                else {
                       pEchoReply = (PICMP_ECHO_REPLY)lpReplyBuffer;
                       totalRTT += pEchoReply->RoundTripTime;
                }
        }
        double dTotalTime = static_cast<double>(totalRTT);
        return totalRTT / NUM_PACKETS;
}
```

Results:

Type	Measured Round Trip Time (ms)	Estimated Round Trip Time (ms)
ICMP Remote	2.235	1.5
ICMP Loopback	0 (<1)	1
TCP Remote	6.568	3
TCP Loopback	0 (<1)	2

Discussion:

After viewing the results, we are proven wrong with our software overheads. This is evident because they seem to have minimal effects on our loopback results. On the other hand, our comparison between ICMP and TCP was based on a predicted factor of 2. Actual results show this to be closer to a factor of 3.

Peak Bandwidth

Predictions:

Our network card speed is 450 mbps, upper bounding our bandwidth (for remote communications). As this is a hardware specification, we estimate an additional 20% overhead may occur.

The above only applies for remote connections. For loopback connections, the hardware does not influence our results and is rather measured in copying data to our message buffer.

Method:

For this experiment we set our TCP server to just receive messages, so we reduce additional overheads when receiving and processing messages on the server side. Note, to reduce variables that would hamper our ability to measure peak bandwidth, we only measure how long it would take to send a message. However, there are design issues that would affect our manual timing of the time it takes to send a message when using TCP sockets. Initial results with this method were surprising, both remote connection bandwidth was similar to loopback results. We cite the reason from socket design: “To optimize performance at the application layer, Winsock copies data buffers from application send calls to a Winsock kernel buffer", and "In most cases, the send completion in the application only indicates the data buffer in an application send call is copied to the Winsock kernel buffer and does not indicate that the data has hit the network medium" [8]. This issue was not apparent in the round trip time experiments because a send coupled with a recv form a blocking pair, waiting for data to be received after sending. To circumvent this, we disable Winsock buffering.

We measure how long it would take to send 4KB of data from client to server in µs.

We sent 30 packets in our tests, and took the shortest time out of the results to calculate and find the peak bandwidth in Mb/s.

Pseudocode:

```
int size = 0;
setsockopt(sock, SOL_SOCKET, SO_SNDBUF, (char *)&size, sizeof(size));
__int64 iLowestTime = MAXINT64;
for (int i = 0; i < NUM_PACKETS; i++)  {// For each packet
       QueryPerformanceFrequency(&freq);
       QueryPerformanceCounter(&start_time_stamp);
       int iSent = send(sock, szSendMessage, strlen(szSendMessage), 0);
       QueryPerformanceCounter(&end_time_stamp);
       if (time_elapsed() < iLowestTime)
              iLowestTime = time_elapsed();
}
double dMinTime = static_cast<double>(iLowestTime);
dPeakBandWidth = (double)(MSG_SIZE) / dMinTime * 8.0; // Mbps
```

Results:

Type	Measured Peak Bandwidth (Mb/s)	Estimated Peak Bandwidth (Mb/s)
TCP Remote	520.833333	360
TCP Loopback	1953.125	>450

Discussion:

Our results seem to be incorrect. The send socket function is sending a send completion prematurely to our measuring application, inflating our measurements. We've investigated further and was unable to find out how to receive indication of our data being sent through our network card. We could have tried using a send and recv blocking pair using an extraneous minimal ACK packet, but we believe this would move us farther away from measuring the peak bandwidth.

Connection Overhead

Predictions:

A TCP remote connection setup requires the 3-way handshake, which involves communicating with 3 packets between client and server. When closing a connection, 2 pairs of FIN-ACK messages are used to notify the teardown of the TCP session. A minimum of 5 packets are needed to simulate the setup and teardown of a TCP session.

We base our predictions on a predicted bandwidth of 360 mbps for remote connections and 1900 mbps for local connections. Given that we have a minimum of 1000 bytes to transmit, it would take, in terms of hardware overhead, a setup and teardown would take 0.02 ms in the remote case, and 0.004 ms in the loopback case. Software overhead will be dominant in our measurements due to connection being done on a blocking socket (which is separate from the send issue analyzed in peak bandwidth). We estimate the software difference between remote and loopback as a factor of 3 to account for translating the IP address to the target machine's descriptor.

Method:

For this experiment we set our TCP server to just receive messages. However, this behavior should have no effect on the connection and teardown behaviors.

We measure how long it takes to establish a TCP session between client and server, and close it immediately afterwards. We perform 50 connect setups and teardowns, and take the average time.

Pseudocode:

```
__int64 iTotalTimeElapsed = 0;
for (int i = 0; i < NUM_CONNECTS; i++) {
       SOCKET sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
       QueryPerformanceFrequency(&freq);
       QueryPerformanceCounter(&start_time_stamp);
       if (connect(sock, (struct sockaddr*)&sockaddr, sizeof(sockaddr)) == -1) {
              return -1;
       }
       else { //successfully connected
       }
       closesocket(sock);
       QueryPerformanceCounter(&end_time_stamp);
       iTotalTimeElapsed += time_elapsed();
}
return iTotalTimeElapsed / NUM_CONNECTS;
```

Results:

Type	Measured Connect Overhead (ms)	Estimated Connect Overhead (ms)
TCP Remote	503.36	6
TCP Loopback	90.54	18

Discussion:

The results turn out to be much higher than expected. The software overheads associated with socket connect and close must involve much more complex instructions that we did not account for. We did not consider how long it may take to communicate and verify ports for client and server.

File System

Size of file Cache

Predictions:

Our research indicated that the file cache size on Windows is set dynamically by the operating system at run time. The set value could range from close-to-0 to 50% of the available physical memory [9,10]. As our available physical memory was 8 GB, the cache could be as large as 4 GB. However we predicted that this size should not exceed 1 GB under ordinary usage which might require dedicating portions of the RAM to non-file-IO applications.

<u>Method:</u>

We identified the file block size for our file system to be 4096 Bytes using the *fsutil fsinfo ntfsinfo C:* command and examining the *Bytes Per Cluster*entry. We then identified the file cache size through two "passes" of our algorithm. The initial pass was used for identifying coarsely the interval of the cache. The final pass was used to provide further resolution.

As we predicted the file cache size to be no more than 1 GB, our initial pass created files in exponentially increasing file from 2^0 to 2^{17} blocks in size. We then timed the time it took to sequentially read each of these files from disk. In getting a handle to the file, we used the *CreateFile* API with the FILE_ATTRIBUTE_NORMAL indicating to use the file cache while reading this file. Windows provides a file attribute to hint to the OS to optimize the file cache size for sequential IO, FILE_FLAG_SEQUENTIAL_SCAN. While we considered using this attribute, we chose not too as we felt this would give us a value closer to the maximum and not representative number. To read the files we used the windows *ReadFile* API.

Performing the initial pass allowed us to indentify the file cache size to be between 2^{16} and 2^{17} blocks. We thus performed the final pass for files in that region with file size increasing linearly in steps of 20000 blocks. Timing of the file read time was performed identically to the initial pass.

System call or utility program to determine this metric as a sanity check:

Windows provides the performance counter Memory/Cached Bytes to indicate the file cache size allocated by the operating system [9]. This performance counter can be read through system calls into the registry. To sanity check our measurements, we report the value for this counter after every file read made by our bench mark program. The reported value is the average of 3 consecutive reads.

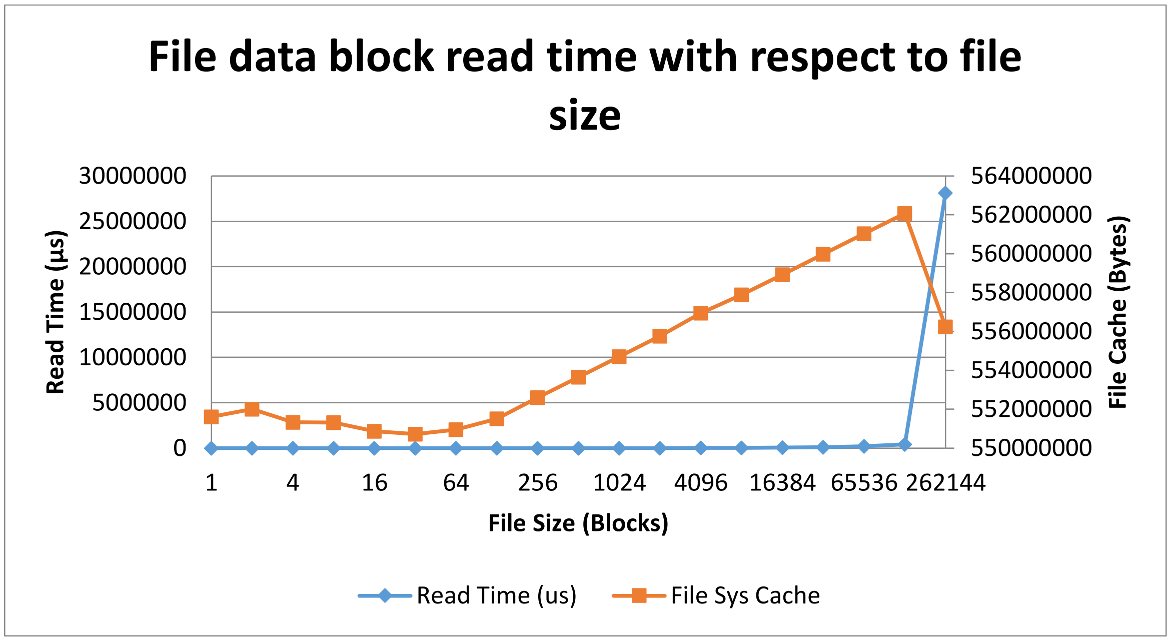
<u>Psuedocode:</u>

```
Given m files of size n blocks:
For i=1,…,m
    Get an access handle for normal (cached) reading and writing to the file.
End for
For i=1,…,m
    Allocate memory for reading file.
    Start timing.
    Read file.
    Stop timing.
    Output elapsed time.
End for
Repeat 10 times and average results.
```

<u>Results:</u>

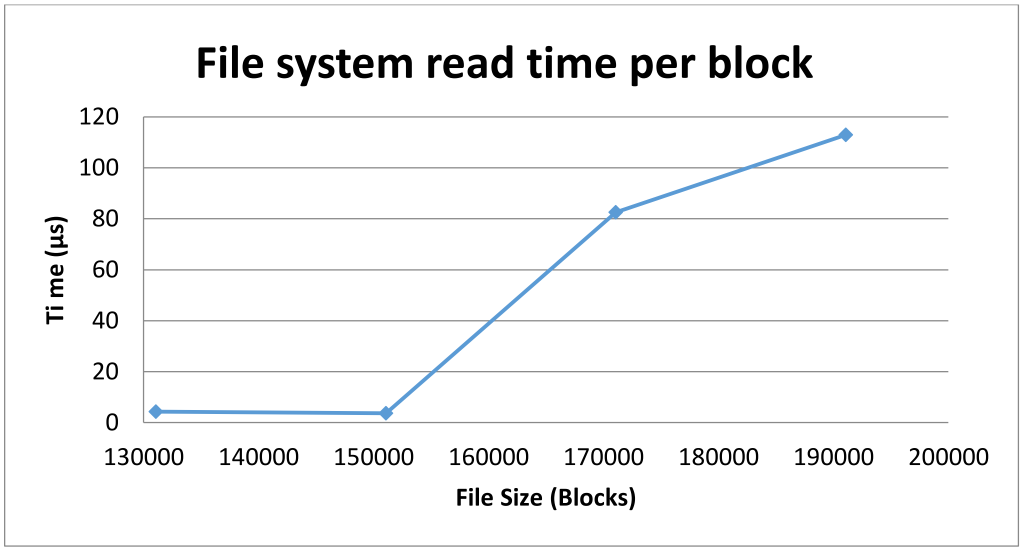
Initial Pass:

File Size (Blocks)	File Size (Bytes)	Average Read Time (µs)	Average Read Time per Block (µs)	File Sys. Cache Size (Bytes)
1	4.10E+03	37	37	5.52E+08
2	8.19E+03	36	18	5.52E+08
4	1.64E+04	42	10.5	5.51E+08
8	3.28E+04	53	6.625	5.51E+08
16	6.55E+04	52	3.25	5.51E+08
32	1.31E+05	73	2.28125	5.51E+08
64	2.62E+05	113	1.765625	5.51E+08
128	5.24E+05	403	3.1484375	5.52E+08
256	1.05E+06	760	2.96875	5.53E+08
512	2.10E+06	2070	4.04296875	5.54E+08
1024	4.19E+06	3032	2.9609375	5.55E+08
2048	8.39E+06	6007	2.933105469	5.56E+08
4096	1.68E+07	11940	2.915039063	5.57E+08
8192	3.36E+07	30941	3.776977539	5.58E+08
16384	6.71E+07	53356	3.256591797	5.59E+08
32768	1.34E+08	107813	3.29019165	5.6E+08
65536	2.68E+08	217823	3.32371521	5.61E+08
131072	5.36E+08	415363	3.168968201	5.62E+08
262144	1.07E+09	28148293	107.3772163	5.56E+08
	Average:		11.71498811	5.55E+08



Final Pass:

File Size (Blocks)	File Size (Bytes)	Average Read Time (µs)	Average Read Time per Block (µs)	File Sys. Cache Size (Bytes)
131072	536870912	560693	4.277748108	576972117
151072	618790912	551570	3.651040563	576083285
171072	700710912	14124161	82.56266952	576645802
191072	782630912	21566895	112.8731316	546082816
	Average:		22.19337746	560021396



Discussion:

Performing the initial pass allowed us to indentify the file cache size to be between 2^{16} and 2^{17} blocks as between these two sizes we see the average read time per block increase 5-fold indicating that the system is no longer able to cache the entire read. As our disk has no on-disk cache, the increase in read time can only be attributed to file system cache size being exceeded. Performing the second pass allowed us to successfully identify this value to be at least 151072 blocks or 618 MB. After this file size we see the average read time per block begin to climb. The cache size returned by the operating system after this read is 576 MB or 7% less indicating that our measurement is close to the actual value.

We expected the cache size reported by the operating system to be higher, not lower than our measurement. This is because the file cache would also be utilized by other resident operating system code and device drivers as well as performing IO for our benchmark. The reported cache size reported back could be lower due to timing delays with fetching this value. As we averaged 3 values with some delay between each measurement, the operating system could have optimized the cache size down by the time we have finished fetching these values. Additionally the file cache may not need to cache the entire file, just the majority of it.

File read time

Predictions:

While we were unable to find the specification for our specific hardware, competitive benchmarks around the same time period indicate seek times of 100 µs and speeds of approximately 200 MB/s. Thus we predict that accessing a single block of data should take 21 µs with no seek time overhead and 121 µs with seek time overhead. Software overhead would add an approximate 2 µs of overhead due to having to make a system call and crossing the user-mode and kernel boundary. Thus our expected read time for sequential reads with minimal seek time between reads is 23 µs per block and the expected read time for random access is 123 µs per block to account for inter-block seeks.

Method:

Te measure sequential read time we read files of exponentially increasing file size from 2^0 to 2^{16} blocks. We then timed the time it took to sequentially read each of these files from disk. In getting a handle to the file, we used the *CreateFile* API with the FILE_FLAG_NO_BUFFERING to disable the use of the file cache in reading and writing to this file [10]. To read the files we used the windows *ReadFile* API. Our disk has no on-disk cache and thus disabling IO buffering by the operating system should result in clean measurements.

To measure random read time to the same files we used the C++ input file stream object, std::ifstream, as we found no operating system API to support this. The std::ifstream::seekg function allowed us to skip to a specific byte in the file so that we could read in the file blocks in a random order. We used the C++ random number generator function, *rand*, to create a random block index. As this added measurement overhead we ran our code with and without reading the files and subtracted the two to obtain a corrected measurement.

Psuedocode:

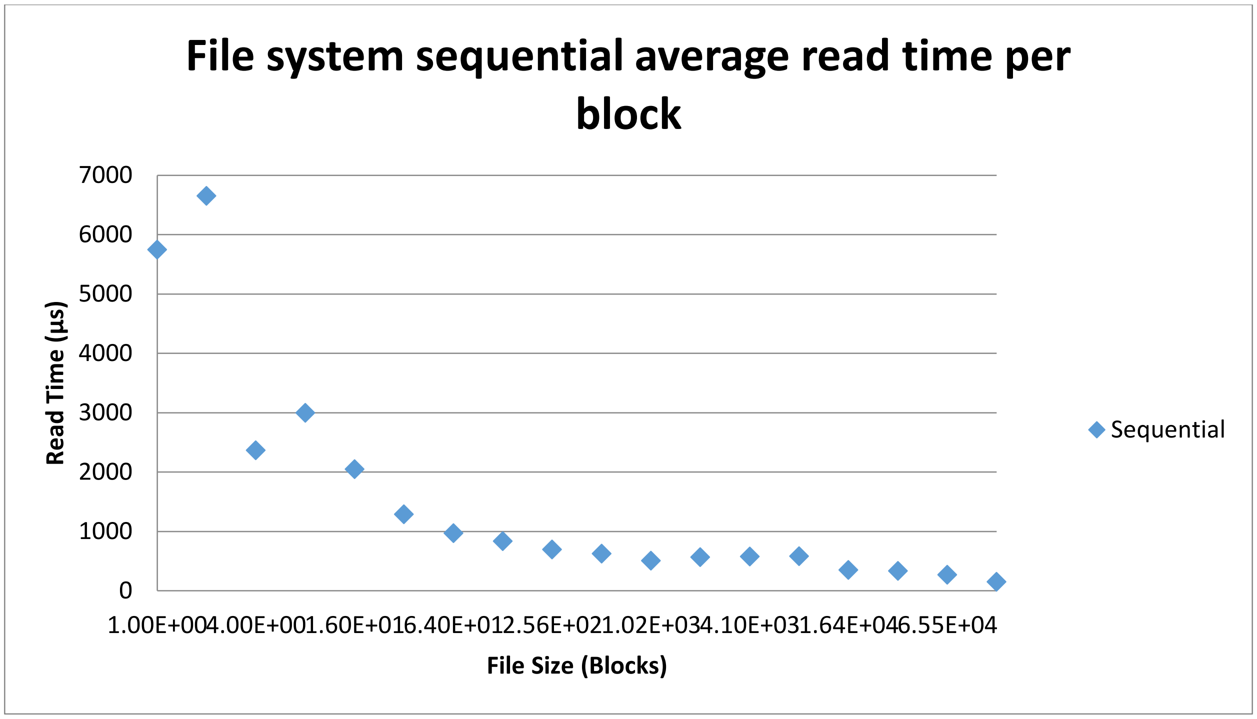
```
Given m files of size n blocks:
For i=1,…,m
    Get an access handle for non-cached reading and writing to the file.
End for
// Sequential read
For i=1,…,m
    Allocate memory for reading file.
    Start timing.
    Read file.
    Stop timing.
    Output elapsed time.
End for
// Random read
For i=1,…,m
    Close handle.
    Allocate memory for reading file.
    Start timing.
    For j=1,…,n
        Obtain random index k ranging from 1,…,n
        Read file block from file m starting from index k
    End for.
    Stop timing.
    Output elapsed time.
```

```
End for
Repeat 10 times and average results.
```

<u>Results:</u>

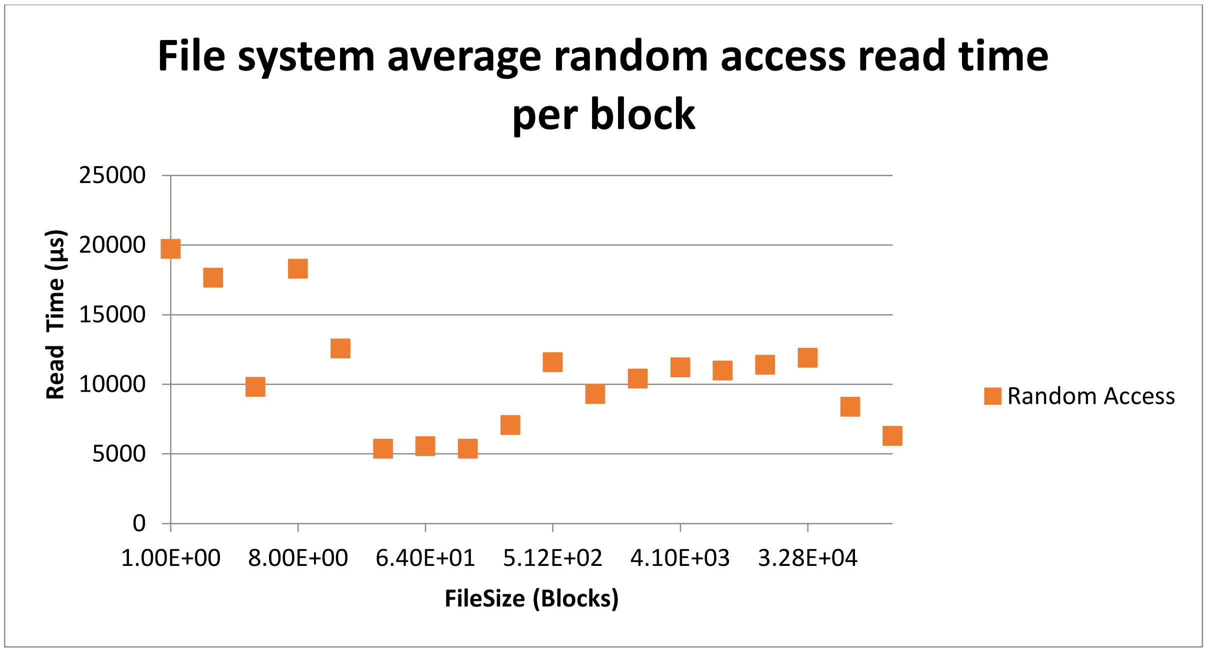
Sequential Read Time:

File Size (Blocks)	File Size (Bytes)	Average Read Time (us)	Average Read Time per Block (us)
1	4.10E+03	5746	5746
2	8.19E+03	13305	6653
4	1.64E+04	9466	2367
8	3.28E+04	23988	2999
16	6.55E+04	32819	2051
32	1.31E+05	41312	1291
64	2.62E+05	61956	968
128	5.24E+05	106857	835
256	1.05E+06	177717	694
512	2.10E+06	321587	628
1024	4.19E+06	519638	507
2048	8.39E+06	1158346	566
4096	1.68E+07	2361241	576
8192	3.36E+07	4783149	584
16384	6.71E+07	5754986	351
32768	1.34E+08	10968077	335
65536	2.68E+08	17558544	268
131072	5.37E+08	20015730	153
	Average:		1532

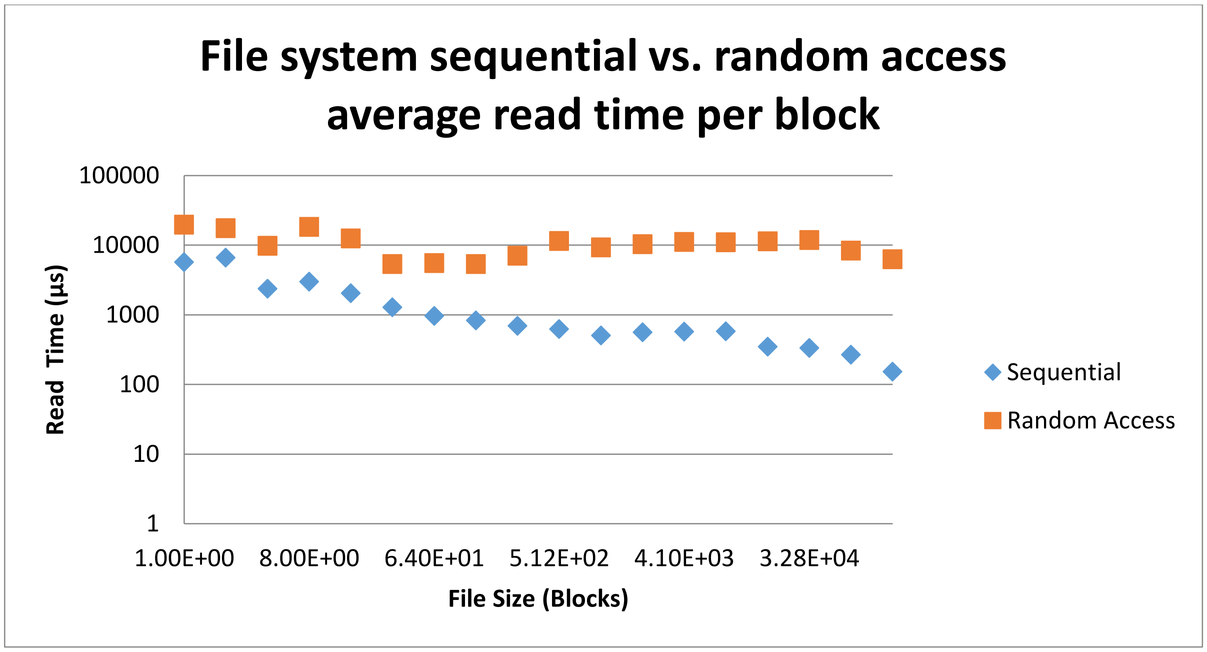


Random Access Read Time:

File Size (Blocks)	File Size (Bytes)	Corrected Average Read Time (µs)	Corrected Average Read Time per Block (µs)
1	4.10E+03	19738	19738
2	8.19E+03	35320	17660
4	1.64E+04	39243	9811
8	3.28E+04	146407	18301
16	6.55E+04	201046	12565
32	1.31E+05	172131	5379
64	2.62E+05	356155	5565
128	5.24E+05	688095	5376
256	1.05E+06	1812916	7082
512	2.10E+06	5928087	11578
1024	4.19E+06	9540053	9316
2048	8.39E+06	21336457	10418
4096	1.68E+07	45924989	11212
8192	3.36E+07	90056897	10993
16384	6.71E+07	186982633	11413
32768	1.34E+08	390508851	11917
65536	2.68E+08	550079238	8394
131072	5.37E+08	826343672	6305
	Average:		10723



Comparison:



<u>Discussion:</u>

We measured sequential read time to be 1532 µs and random access time to be 10723 µs on average, 7 fold slower. Both of these times are significantly lower than predicted indicating that our disk read bandwidth was much slower than specified. This could be to the fact that the test machine has been in use for some time and performance has decreased. This could also be due to the fact that the software overhead added by the methods used is significantly more than expected. Additional overhead is corroborated by the fact that average read times per block for files only a few blocks in length were significantly higher than for files several hundred blocks in length for both sequential and random access. This points to a fixed overhead for accessing a file.

We noted that sequential access times per block decreased with file size. This indicates that the files system is more efficient at accessing sequential bytes with little seek overhead. We also noted that there were 3 “levels” of access times present with transition at 4 and 8192 file blocks. This could be due to batching of reads for lower file sizes and not-true sequential file access for larger file sizes.

Access times for random access did not change nearly as significantly as for sequential access. We did see a drop in access times for files 32 blocks in size and a subsequent increase for files 512 blocks in size. For an SDD seek times should not increase with respect to physical location. Thus these results point at a fixed overhead or batching of smaller reads. As we rely on std::ifstream to provide the desired behavior, we cannot guarantee that the random reads were not using the file cache in all cases. This object is also likely to add additional computational overhead. Reads from larger files are more likely to be from outside of cache for larger files and this can explain the slightly larger read time.

How "sequential" access might not be sequential:

We assume that for the sequential file read operation the file blocks are physically spaced on the disk in sequential order such that there is no seek time overhead between the reads. This may not be the case, especially for very large files, which may have been split into multiple disk regions upon writing. As we wrote the files in sequential order and our test machine’s disk had not been previously been used at more than 13% capacity (and thus should not be significantly fragmented), we conclude that overall we could guarantee a decrease in seeks between consecutive blocks, even for the larger test files. Our test results indicate that the access times per block decreased significantly with file size. Thus these results corroborate this conclusion.

Remote file read time

Predictions:

Our network bandwidth for TCP measured to be 520 Mb/s. This is approximately 3-fold slower than our likely SDD read bandwidth of 200 MB/s. To read the data remotely we would need to first request the read, read it from disk on the remote file system, then send it over the network. Thus the net overhead would be at least 4-fold slower.

Method:

For this operation we tested accessing the file system tested in the previous operation from a similar, equivalent device using a VPN. We mounted the remote file system as the test machine’s “U” drive and were able to access it with the same system calls made in the original methods. As all the test files had been previously created during the previous operation benchmark, we did not create a new file if the file was not found and just returned an invalid handle. This allowed for faster diagnosing of issues we encountered during initial setup. As during testing we found the remote file read time to be quite slow, we also decreased the maximum file size that was read back for sequential and random access to be 2^{15} and 2^{9} blocks respectively. Apart from these minor changes we were able to replicate our previous experiment entirely.

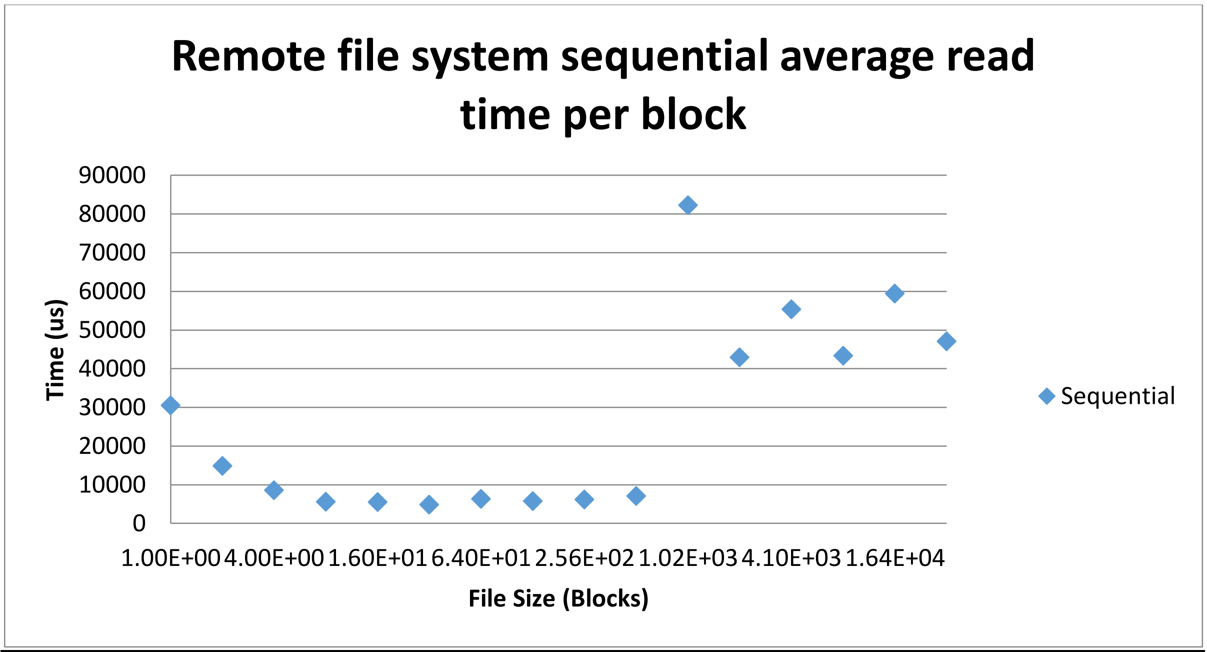
Psuedocode:

Please refer to the Pseudocode section for File read time.

Results:

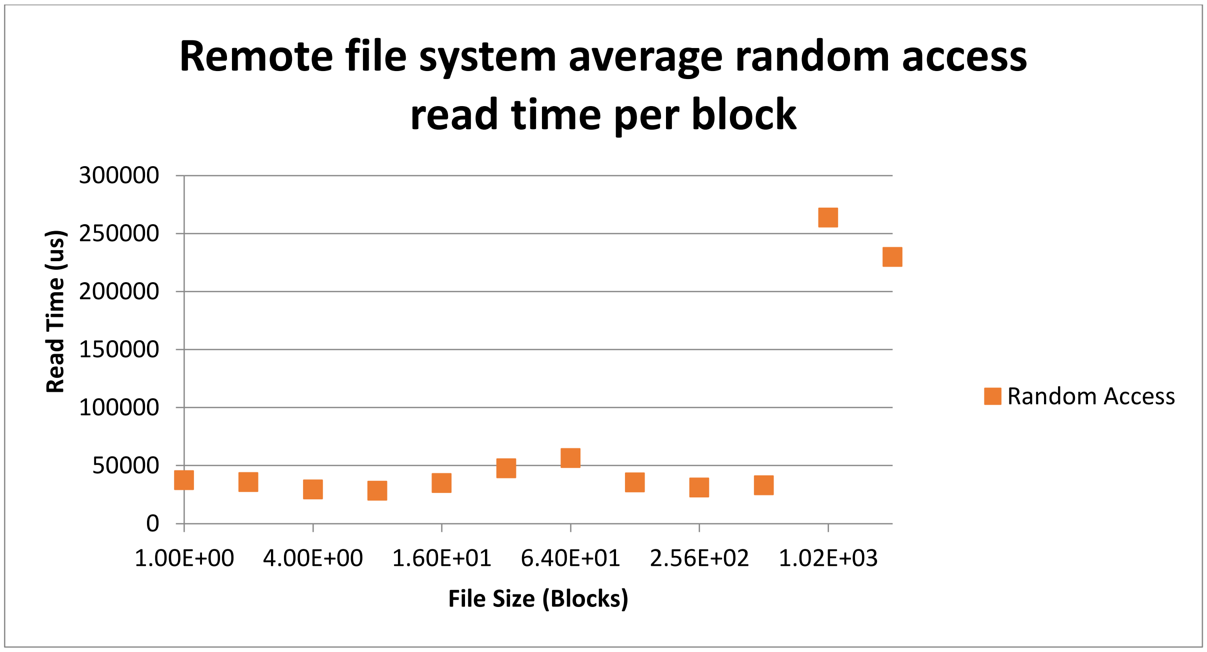
Sequential Read Time:

File Size (Blocks)	File Size (Bytes)	Average Read Time (µs)	Average Read Time per Block (µs)
1	4.10E+03	30544	30544
2	8.19E+03	29787	14894
4	1.64E+04	34220	8555
8	3.28E+04	44575	5572
16	6.55E+04	88402	5525
32	1.31E+05	155801	4869
64	2.62E+05	402972	6296
128	5.24E+05	740472	5785
256	1.05E+06	1578881	6168
512	2.10E+06	3625351	7081
1024	4.19E+06	84233190	82259
2048	8.39E+06	87850831	42896
4096	1.68E+07	226772833	55364
8192	3.36E+07	354839868	43315
16384	6.71E+07	972864088	59379
32768	1.34E+08	1543048200	47090
	Average:		26599

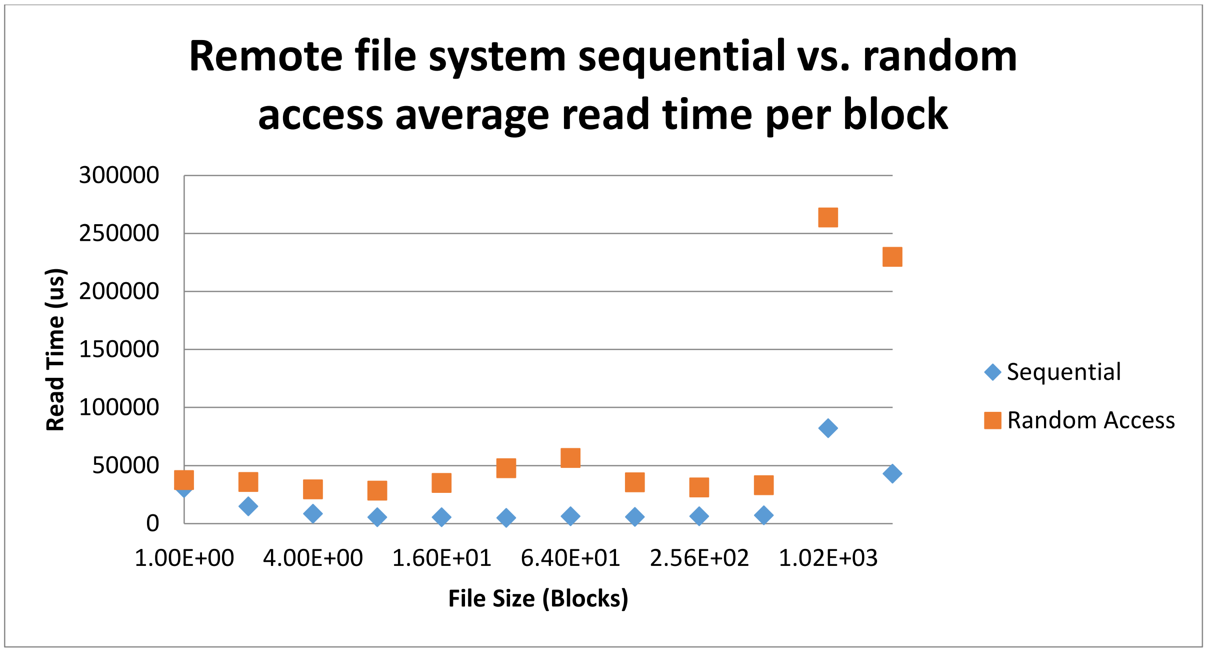


Random Access Read Time:

File Size (Blocks)	File Size (Bytes)	Corrected Average Read Time (µs)	Corrected Average Read Time per Block (µs)
1	4.10E+03	37449	37449
2	8.19E+03	71711	35856
4	1.64E+04	118289	29572
8	3.28E+04	226652	28332
16	6.55E+04	558512	34907
32	1.31E+05	1526642	47708
64	2.62E+05	3614667	56479
128	5.24E+05	4560740	35631
256	1.05E+06	7942778	31026
512	2.10E+06	16943196	33092
1024	4.19E+06	270102990	263772
2048	8.39E+06	470535939	229754
	Average:		71965



Comparison:



<u>Discussion:</u>

Average remote sequential read time per block was 26599 µs and remote random access read time per block was 71965 µs, a 3-fold difference. As sequential read time was faster we were able to read in files of larger size. Thus normalized with respect to file length read this difference is 4-fold.

The results indicate that we were not able to guarantee un-buffered reads as the read times for both sequential and random reads jump significantly for files greater than 512 blocks in size. While we used the same method as for local reads, it is likely that the files are being fetched entirely from the remote file system up to a certain size, despite the file attributes specified. We then perform our read operations on a local equivalent of the file, and thus the random access overhead is not as large as for the local file system.

The "network penalty" for accessing files over the network:

Sequential remote read time per block is 17-fold greater than the local equivalent. Random access time is 7-fold greater. This is even slower than the predicted overhead of 4-fold slower. This is likely due to the added overhead of VPN.

Contention

<u>Predictions:</u>

We predicted that the average read time it took to read a block would increase exponentially with each added process. Results for the second part of this section indicated that it took approximately 6,000 µs to read a file one data block in size. Per our prediction this time would double as the number of processes trying to access different files doubled.

<u>Method:</u>

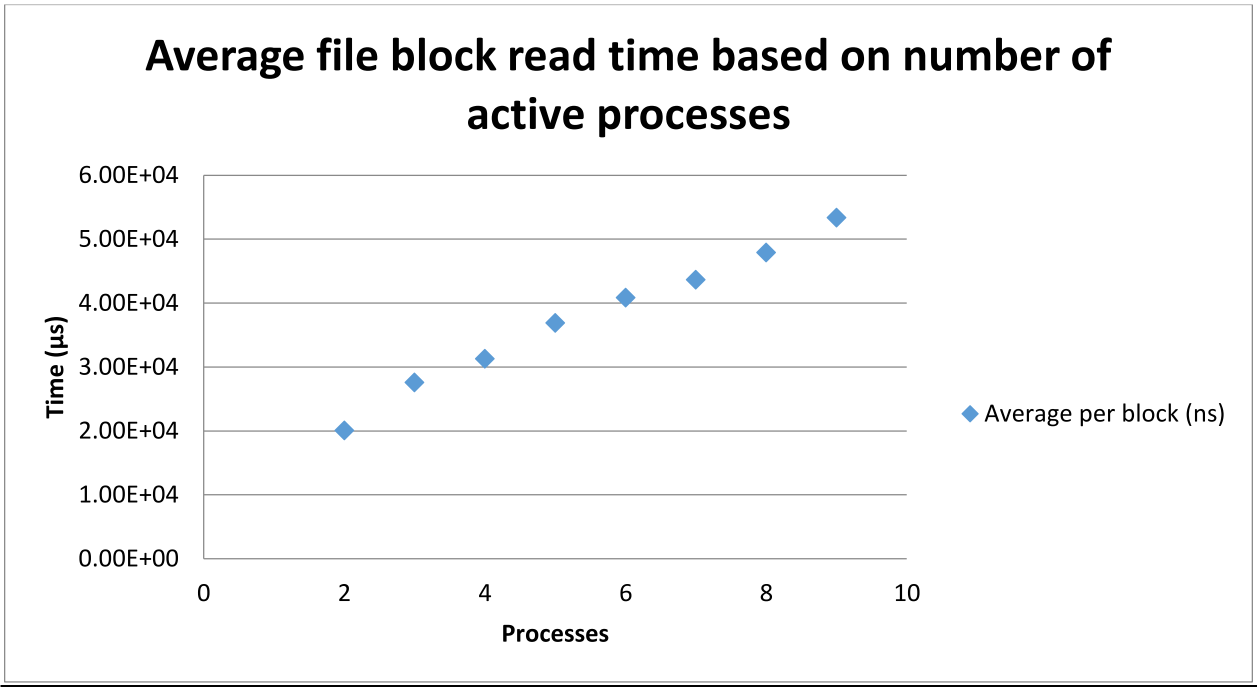
To measure contention we created 1 through 9 test processes with a unique identifier. Each process initially created a set of 100 test data files 2 blocks in length. The process then timed the time it took to sequentially read one block for each file. In getting a handle to the file, we used the *CreateFile* API with the FILE_FLAG_NO_BUFFERING to disable the use of the file cache in reading and writing to this file [10]. To read the files we used the windows *ReadFile* API. The process reported the average for the last 50 files in an output file tagged with its identifier. We introduced a second of sleep before the data was outputted to allow other processes to finish their IO before writing the output file. We did an initial run with 9 active processes to ensure all files were created. We then performed 10 runs with 2,3,…,9 processes being created and then parsed and averaged the results.

Psuedocode:

```
Main process:
    For i=1,…,m where i is an element {2,3,…,9}
        Create process with identifier i
    End for
Each child process with process identifier p:
    For i=1,…,100
        Get an access handle for non-cached reading to the file tagged p and i.
        Create the file if it does not exist.
    End for
    For i=1,…,m
        Allocate memory for reading file.
        Start timing.
        Read 1 block from file.
        Stop timing.
        Update a moving average of length 50 with the elapsed time.
    End for
    Sleep for 1 second.
    Output the moving average to an output file tagged p.
```

Results:

Processes	Average read time per block (us)	Predicted Results (µs)
9	53380	135765
8	47900	96000
7	43685	67882
6	40883	48000
5	36914	33941
4	31288	24000
3	27583	16971
2	20092	12000



Discussion:

Our results indicate that the read time per block increased linearly with the number of processes, and not exponentially as predicted. However this matches our expectation that processes interspersing their reads increases the demands on the file IO and thus read time. It was interesting to note that two processes would take 80 ms on average to read 8 blocks of data whereas 8 processes would take 48 ms, which is still faster. This indicates that for reading single-block file IO multiple processes would still be faster. Our expectation was for this to be this to be reversed.

Measurement error could result from the small number of runs averaged. We also did not explicitly synchronize the child process execution times, but instead discounted the first 50 measured times for each process. This might not have been sufficient in all cases. Additionally it may have been valuable to test files of different length as perhaps inputs of single-block size are an exception due to software overhead involved in initiating the IO.

References

[1] "Game Timing and Multicore Processors." *MSDN*. Web. 28 Jan. 2015.

[2] "QueryPerformanceCounter function." *MSDN*. Web. 28 Jan. 2015.

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

[4] "Reserving and Committing Memory." *MSDN*. Accessed online 18 Feb. 2015 at https://msdn.microsoft.com/en-us/library/windows/desktop/aa366803(v=vs.85).aspx

[5] "TCP." Transfer Control Protocol, 3-way Handshake, Sliding Window. Web. 1 Mar. 2015. <http://www.rhyshaden.com/tcp.htm>.

[6] "ICMP (Internet Control Message Protocol)." Internet Control Message Protocol, ICMP, Ping. Web. 1 Mar. 2015. <http://www.rhyshaden.com/icmp.htm>.

[7] "Using Winsock." *MSDN*. Web. 1 Mar. 2015. <https://msdn.microsoft.com/en-us/library/windows/desktop/ms740632(v=vs.85).aspx>.

[8] "Design Issues - Sending Small Data Segments over TCP with Winsock." Design Issues - Sending Small Data Segments over TCP with Winsock. MSDN. Web. 15 Mar. 2015. <http://support.microsoft.com/en-us/kb/214397>.

[9] "Monitoring the File System Cache." *TechNet*. Accessed online 14 Mar. 2015 at https://technet.microsoft.com/en-us/library/cc757398%28v=ws.10%29.aspx

[10] "File Caching (Windows)." *MSDN.* Accessed online 14 Mar. 2015 at https://msdn.microsoft.com/en-us/library/windows/desktop/aa364218%28v=vs.85%29.aspx

[11] Schoeb, Leah. "Should you believe vendors' jaw-dropping solid-state performance specs?" *Storage* Magazine. January 2013. Accessed online 14 Mar. 2015.