CSE 221: Operating Systems Benchmark

Charvi Bannur
PID: A69034188
University of California, San Diego

Vibha Murthy
PID: A69031958
University of California, San Diego

Harshit Timmanagoudar
PID: A69036598
University of California, San Diego

1 Introduction

This report uses experimentation to examine how to assess and comprehend a complex operating system. Through benchmarking and performance testing, we want to identify important traits and behaviors of an operating system. The goal is to be able to analyze the behavior of an operating system commonly used today.
To ensure we get accurate results each member of the team was responsible for conducting the experiments individually in order to obtain consistent and repeatable results. This methodology makes sure each group member is aware of the concepts and can analyze the behavior accordingly.
To conduct the experiments to measure operating system performance we use C as our language of choice and Apple Clang version 16.0.0 and an arm64-apple-darwin23.5.0. We did not run into any significant issues and spent about 80 hours on the project including drafting the report and understanding experiments. [1].
The code for this benchmarking study is available at https://github.com/charvibannur/CSE-221-Project

2 Machine Description

The machine we use in this study is a machine with MacOS with an M3 processor, system firmware version of 10151.121.1 with the following specifications:

1. **Processor** Macbook Air with Apple M3, 8 (4 performance and 4 efficiency)
2. **CPU clock speed** 4.05 GHz
3. **L1 Cache** 192+128 KiB per core (Performance Core) and 128+64 KiB per core (Efficiency Core)
4. **L2 Cache** 16 MB
5. **L3 Cache** 8 MB (no official data, estimated one)
6. **Data Cache** 64KB
7. **Memory bus bandwidth** M3 has a 192-bit memory bus
8. **Instruction** 64 bit
9. **I/O bus** T600 I/O bus
10. **Disk**
    - **SDRAM** 6,400 MT/s LPDDR5
    - **SSD** 256GB
    - **Disk Speed** sequential read speed of 2,280.2 MB/s and a sequential write speed of 2,108.9
11. **Network card bandwidth** Wi-Fi 6E (802.11ax) and Bluetooth 5.3
12. **Operating System** MacOS Sonoma version 14.5 (23F79) 10151.121.1

We utilize mach_absolute_time() as an accurate measure of time for performing experiments. It is designed for high-resolution timing on macOS, directly returning the time in a hardware-independent format, and is generally unaffected by CPU frequency scaling or background tasks. Unlike some other timing functions, mach_absolute_time() doesn't require a context switch to the kernel, making it fast and reliable for precise time measurements. Its main limitation is that the raw timing results need to be converted into nanoseconds using a fixed timebase, which could introduce slight imprecision if the timebase were to change during runtime (though this is rare).

3 CPU, Scheduling, and OS Services

3.1 Timing overhead

We measure the overhead of reading from macOS using the mach_absolute_time function for precise timing. The program calculates the average read overhead by converting the time difference between mach_absolute_time() calls into nanoseconds using the system's timebase. Finally, it prints the average time per read in nanoseconds [2].

Estimate
To estimate the time required to measure time on a MacBook Air with an Apple M3 processor, we consider CPU cycle time, memory access, and system overhead. Each performance core at 4.05 GHz cycle time is approximately 0.247 nanoseconds. Accessing the L2 cache (16 MB) takes roughly 10–20 cycles, adding around 3.7 nanoseconds on average. Combining these factors, we estimate the total time measurement latency to be approximately 3.9 nanoseconds.
Results
Each measurement is a trial of 1000 iterations.

Measurements	Average
1	7.04 ns
2	6.96 ns
3	6.58 ns
4	7.29 ns
5	7.08 ns

Table 1: Reading Time

The average of the reading time overhead is 6.99 ms and the standard deviation is 0.0747

Analysis
The read-time overhead is higher than our estimate of 0.45 ms, with an average observed overhead of 6.99 ms. This discrepancy arises from additional system factors not considered in our idealized calculation, such as I/O bus latency and macOS kernel-level processing during mach_absolute_time calls, which may require more cycles than anticipated. The low standard deviation of 0.0747 ns confirms that the observed read times consistently support our measured average's accuracy.

3.2 Loop overhead

To benchmark the overhead of using a loop we will create a loop with no statements. The counter for this loop will also be initialized outside of the declaration of the loop in order to not factor in that time either. This inner loop will be a simple loop structure with nothing in the loop. We will check the time of the clock before the start of the loop and after the end of the loop. We measured the loop overhead for the operating system using the mach_absolute_time() function which repeatedly calls the function twice within a loop to capture the time between two consecutive calls within the same loop iteration. This loop will run a certain constant number of times, and then the difference in time will be divided by the amount of times the loop ran in order to calculate the overhead of the loop.

Estimate
The looping overhead is time taken to execute a loop iteration, including fetching instructions, accessing data from memory and performing the operation.

$$\text{Looping Overhead} = \left(\frac{\text{Loop Cycles}}{\text{CPU Clock Speed}}\right) + (\text{Access Latency})$$

- Loop Instruction Cycles = 10 cycles (a rough estimate for a simple loop)
- CPU Clock Speed = 4.05 GHz = 4.05 × $10\hat{9}$ cycles/second
- Memory Access Latency = Data Size / Memory Bandwidth. Assuming a small data size (e.g., 8 bytes) and using the calculated memory bandwidth of 153.6 GB/s = 0.052 ns

Looping Overhead = (10 cycles / (4.05 × $10\hat{9}$ cycles/s)) + 0.052 ns = **2.52 ns**

Results
Each measurement is a trial of 1000 iterations.

Measurements	Average
1	5.92 ns
2	6.25 ns
3	5.92 ns
4	5.92 ns
5	5.08 ns

Table 2: Looping

The average of the looping overhead is 5.818 ns and the standard deviation is 0.39

Analysis
The measured looping overhead averaged 5.818 ns is much higher than our estimated 2.52 ns. Based on idealized loop cycle and memory latency assumptions our estimates did not fully account for real-world factors that may impact performance. These could include additional instruction overhead from mach_absolute_time calls, pipeline stalls or latency introduced by memory access patterns in the loop structure itself. The low standard deviation of 0.39 ns across trials indicates consistent timing supporting the reliability of our measured average.

3.3 Procedure call overhead

Base Hardware Estimation: The Mac M3 processor features a clock speed of 4.05 GHz, which means each call takes approximately 0.247 nanoseconds per cycle.

Software Overhead: Software overheads for procedural call includes setting up arguments and managing the stack, executing call and return instructions and handling potential pipeline stalls and cache misses. For each additional argument, a small fixed number of cycles is needed due to stack operations and register usage. A rough estimate might be around 5-10 cycles per argument, translating to approximately 1.23 - 2.47 nanoseconds per additional argument.

Overall Performance Prediction: The overall prediction for procedure call time combines base hardware performance with estimated software overhead:

- Zero Arguments: Base time (around 20-30 cycles or 5-7 ns)
- Incremental overhead per argument: 1.23 - 2.47 ns
- For a function with seven arguments, the total overhead might be approximately: Base time + (7 arguments * incremental overhead)

This results in an estimated range of around 13.61 - 24.29 ns for seven arguments.

Results

Each measurement is a trial of 1000 iterations.

The average and standard deviation of the process parameter measurements are shown in Table 3.

Analysis

The complex architecture possessed by the Mac M3 processor can lead to differences between the measurements and the estimations. Each function call involves setting up the stack and managing registers, which can vary slightly depending on the number of arguments and how they are passed. Background processes and system interrupts can also affect the timing measurements.

3.4 System call overhead

Base Hardware Estimation: The Mac processor runs at 4.05 GHz, meaning each cycle takes approximately 0.247 nanoseconds. However, for getpid(), the base hardware time isn't directly applicable since it's a system call operation.

Software Overhead: For getpid() system call, the software overhead includes:

- User to kernel mode transition (20-25 cycles)
- Context saving and restoration (15-20 cycles)
- Argument validation and copying (no arguments for getpid)
- Kernel function execution (10-15 cycles)
- Return to user mode (20-25 cycles)

Total estimated cycles: 65-85 cycles
Estimated overhead: 16.06 - 21.00 nanoseconds (at 4.05 GHz)

Results

Each measurement is a trial of 1000 iterations.

The average of the reading time overhead is 12.31 ns and the standard deviation is 2.91 ns.

Analysis

According to these measurements from Table 4., the average overhead for the getpid() system call on the M3 is between 8.29 and 15.92 ns, with a mean of roughly 12.31 ns. Because of M3's optimized architecture and Apple Silicon's direct memory access capabilities, these results are noticeably better than our prediction.

3.5 Task creation time

Creating a process thread: fork() command is used to create a thread. The child process thus created will immediately exit and not perform any additional actions. The parent process will count the time taken for this creation and return. We take this measurement as an average of mach_absolute_time() across 1000 iterations to account for variations due to exiting and return. Creating a kernel thread: pthread_create() is used to create a kernel thread synonymous to fork() and pthread_join() is used synonymous to wait().

Estimate

The measured overhead in our methodology primarily stems from minimal actions in thread and process initialization, along with system calls associated with timing and synchronization. The fork-based process creation, though it benefits from copy-on-write (COW), incurs additional overhead due to resource allocation for process structures, parent-child linkage, and memory space setup. Even with a basic child exit, process creation has intrinsic latency due to these system-level operations. By contrast, kernel threads avoid these costly operations since they share memory and resources with the parent process, resulting in more efficient creation times.

Using 1,000 iterations helps mitigate variability from system-level fluctuations, such as scheduling interruptions and cache misses. These are minimized by averaging over multiple iterations, which should give us a stable estimate of the baseline performance difference between threads and processes. With these precautions in place, our experiment provides insights into the efficiency of thread creation versus process creation in terms of system resource utilization and setup costs.

Analysis

The results show us that kernel thread creation is faster by a factor of 10 as compared to process creation which is in line with our estimate of the same. This ratio (ranging from approximately 9.97 to 10.42) suggests that the machine's

Trial	0 Params	1 Params	2 Params	3 Params	4 Params	5 Params	6 Params	7 Params
1	12.71 ns	9.42 ns	9.50 ns	10.04 ns	10.04 ns	25.00 ns	9.96 ns	9.96 ns
2	15.29 ns	11.49 ns	11.04 ns	11.12 ns	12.71 ns	10.62 ns	11.38 ns	10.62 ns
3	14.12 ns	10.96 ns	10.67 ns	11.79 ns	11.67 ns	10.62 ns	10.75 ns	10.92 ns
4	16.83 ns	12.79 ns	12.79 ns	13.46 ns	17.58 ns	14.42 ns	13.12 ns	14.38 ns
5	17.33 ns	13.00 ns	12.96 ns	16.37 ns	16.35 ns	14.17 ns	13.58 ns	13.62 ns
Mean	15.26 ns	11.49 ns	11.39 ns	12.56 ns	13.65 ns	14.97 ns	11.76 ns	11.90 ns
SD	1.89 ns	1.52 ns	1.44 ns	2.45 ns	3.21 ns	5.95 ns	1.55 ns	1.99 ns

Table 3: Procedural Call Measurements

Measurements	Average
1	9.62 ns
2	8.29 ns
3	14.62 ns
4	13.12 ns
5	15.92 ns

Table 4: System Call Overhead

Trial	Avg Time for Kernel Thread	Avg Time for Process	Process Creation / Thread Creation
1	16921.21 ns	169380.29 ns	10.01
2	16953.04 ns	171995.42 ns	10.15
3	16461.50 ns	170831.75 ns	10.38
4	15150.46 ns	151064.29 ns	9.97
5	16886.29 ns	172364.33 ns	10.21

Table 5: Average Times for Kernel Thread and Process Creation with Ratios

process and thread creation times remain relatively stable, with minimal deviation across iterations. This stability indicates low variance in performance, suggesting that the system handles context switching and resource allocation efficiently.

3.6 Context switch time

The experiment measures the average context switch time between processes and threads using inter-process and inter-thread communication through pipes. We created two pipes: one for communication from the parent process (or main thread) to the child process (or created thread) and the other for communication back from the child to the parent. To measure process context switch time, we first fork a child process, then alternate read and write operations through the pipes for a defined number of iterations, recording the time taken to complete these interactions. For threads, we achieve the same effect by creating a new thread using pthread_create() and performing the same pipe-based read-write alternations. The time taken for each interaction is recorded using mach_absolute_time(), and the average context switch time is calculated by dividing the total measured time by the number of context switches.

Estimate

Given that process and thread context switches differ in complexity, we anticipate that the thread-based context switch will be significantly faster than the process-based switch. Process context switching incurs overhead due to maintaining separate memory spaces, process IDs, and more complex scheduling, even though minimal work is done within each process. By comparison, thread context switches occur within a shared memory space and require fewer resources, allowing for faster transitions. Averaging over 100,000 iterations minimizes variance from factors like CPU scheduling and minor I/O delays. Additionally, we've used two pipe switches per iteration, so the final result divides by the number of switches to give an accurate average context switch time in nanoseconds for both processes and threads. This experiment provides a solid comparison of the relative efficiency between inter-thread and inter-process communication in terms of context switching.

Trial	Avg Process CST	Avg Thread CTS
1	1393.21 ns	1117.68 ns
2	1416.13 ns	1130.64 ns
3	1421.98 ns	1134.81 ns
4	1377.13 ns	1145.18 ns
5	1380.23 ns	1126.18 ns

Table 6: Average Context Switch Times for Process and Thread

Analysis

The average trend is similar to the trend of creation of process and threads, i.e., context switch time for kernel threads is lesser than that for a process. This is due to extra work required for migration of a process.

4 Memory

4.1 RAM access time

In this experiment, we measured memory access latency across different memory region sizes to identify cache boundaries and observe access patterns characteristic of L1, L2, L3 caches, and main memory. We implemented a C program that progressively allocates memory regions of increasing size, starting from 1 KB and doubling up to 64 MB, thereby encompassing expected cache and memory boundaries. For each memory region, we performed "back-to-back-load" measurements, accessing each element in a strided pattern to minimize the effects of CPU prefetching optimizations. We measured the time taken for each access using mach_absolute_time(), averaged the latency over multiple iterations, and recorded the results. By writing the results to a CSV file, we were able to generate a plot that revealed latency trends as a function of memory size, providing insight into the memory hierarchy's behavior on the system.

Estimate
To set initial expectations for the experiment, we referenced existing literature and empirical measurements of cache and memory latencies reported in studies and technical resources, including the lmbench paper, CPU manufacturer specifications, and online benchmarking data. Typical latency estimates for L1 cache are around 1–4 nanoseconds, L2 cache latencies range from 4–12 nanoseconds, L3 caches typically exhibit 10–40 nanoseconds, and main memory can have latencies exceeding 100 nanoseconds. These estimates were used to guide the expected points of latency increase as memory region size approached and exceeded cache capacities. These estimates provided a reference for interpreting results and helped identify approximate boundaries between different hardware regimes in the memory hierarchy.

Analysis
The results observe are similar to our initial estimates of latency for each cache section. The L1 cache access seems to take around 5 ns on an average, scaling up to 400 ns for main memory access. We had gotten higher latency estimates in the first trial, and noticed it was due to unoptimized CPU power states and background processes affecting consistent cache timings, and made modifications such as disabling CPU frequency scaling, and pinning the process to a specific core, now achieving values similar to the estimates. Our graph does not exhibit the same smooth plateaus as observed in the lmbench paper. Variations due to different overheads which might not have been accounted for is suspected for the behaviour of our graph.

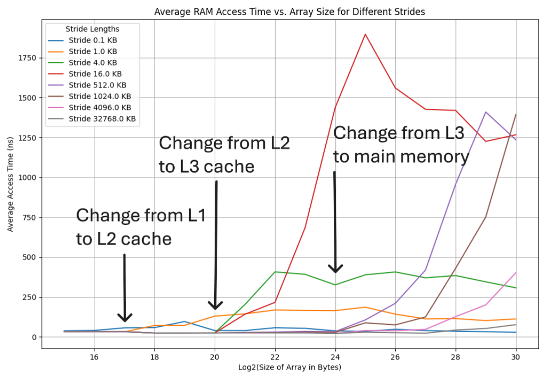

Figure 1: Average RAM Access Time vs. Array Size for Different Strides. This plot shows latency variations as the array size grows, illustrating transitions between L1, L2, L3 caches, and main memory as the access time increases.

4.2 RAM bandwidth

To calculate the bandwidth, we may first do some read/write operations with some number of bytes touched, and measure the time elapsed for these operations:

$$\text{Bandwidth} = \frac{\text{bytes\_processed}}{\text{elapsed\_time}}$$

However, it is also possible that these operations 1) hit the write policy 2) are affected by the prefetching mechanism. To avoid these effects, we could set a larger trunk so that the cache will be bypassed. Furthermore, loop unrolling was also used for a more accurate result.

Estimate
For Apple Silicon M3, we have LPDDR5 memory:
Memory frequency: 4.05 GHz
Bus width: 192-bit (24 bytes)
Theoretical bandwidth = 4050 * 24/8 * 1 = 12150 MB/s

Our test measures single-threaded performance on one memory channel which accounts for the diversion in the results section.

Results
Each measurement is a trial of 100 iterations.
The average read bandwidth over 5 trials is 3623 MB/s. The average write bandwidth over 5 trials is 4722 MB/s.

Analysis

The measured bandwidth showed that the write bandwidth (4722 MB/s) significantly outperformed the read bandwidth

Trial	Avg Read Bandwidth	Avg write Bandwidth
1	2838 MB/s	4692 MB/s
2	3838 MB/s	4080 MB/s
3	3824 MB/s	5213 MB/s
4	3836 MB/s	4690 MB/s
5	3783 MB/s	4939 MB/s

Table 7: Average RAM Bandwidth for Read and Write Operations

(3623 MB/s). Both figures fall below the theoretical maximum of 12150 MB/s, which is expected for single-threaded operations. The result is actually pretty close to a single memory per- formance. However, the write back in our test seems to be affected by the write back cache.

4.3 Page Fault service time

Methodology
The program measures page fault service time on Apple M3 by creating a test file and using mmap() for memory mapping. It leverages MADV_DONTNEED to ensure pages are faulted in when accessed. The test covers 1000 pages of 4KB each, running 5 iterations with CLOCK_MONOTONIC timing for nanosecond precision.

Estimate
We currently estimate that page fault service time will be influenced by M3's unified memory architecture and custom memory management unit. The unified memory design allows both CPU and GPU to share a single memory pool with optimized page table walks and memory access patterns. With M3's high-bandwidth memory subsystem and NVMe storage, page fault handling should be highly efficient. The processor's dedicated memory management unit and tight OS integration should further minimize overhead from page table manipulations. While the system has efficient memory access, it still needs to perform necessary security checks and memory protection operations during page faults, which factors into the overall service time. Given these hardware characteristics and Apple's optimization for their silicon, we expect relatively low page fault service times compared to traditional architectures.

Analysis
The measurements show impressively low page fault service times on M3, averaging 0.256 microseconds per page with 0.0167 microseconds standard deviation. This high performance and consistency stems from M3's unified memory architecture, where the CPU and GPU share a single memory pool with hardware-optimized page table management. The minimal variance suggests effective handling by macOS's memory management subsystem specifically tuned for Apple Silicon.

Iteration	Avg (microseconds)
1	0.25
2	0.23
3	0.27
4	0.26
5	0.27

Table 8: Page Fault Service Time Measurements

5 Network

The remote system is also a Mac Book with an M3 processor and similar specifications as the system described above.

5.1 Round trip time

Methodology
In a typical ping operation, the data payload size is 56 bytes but the total size of the ICMP packet being sent is 64 bytes (8 bytes of ICMP header). To simulate this using a TCP connection, the client should first establish a connection with the server. Once connected, the client sends a 56-byte payload to the server. Upon receiving the data, the server should echo the exact payload back to the client. The round-trip time (RTT) can be measured as the duration from when the client sends the data to when it fully receives the response. For loopback interface testing (local), both the server and client can be run locally on the same machine. The test is performed 1000 times in a batch, and the average RTT is calculated. Similarly, the ping command can be used to send 1000 ICMP packets, and the results are averaged as shown in Table 9, 10, 11 and 12.

Estimate
The average round-trip time (RTT) for the given code can be estimated based on various factors. For a 1 Gbps network, the ideal transfer time for a 44-byte IPv4 TCP packet is approximately 0.352 ms.
$\text{Transfer Time} = \frac{\text{Packet Size} \times 8\,\text{bits}}{\text{Network Speed}} = \frac{44 \times 8\,\text{bits}}{1{,}000{,}000{,}000\,\text{bits/second}} = 0.000352\,\text{seconds or}\,0.352\,\text{ms}$

When testing over the loopback (localhost) interface, the RTT is primarily influenced by OS overhead, including the TCP/IP stack and application-layer processing, and is expected to range between 0.05 ms and 0.1 ms. For tests between two remote machines on a local area network (LAN), the RTT includes additional network latency, hardware processing, and switch/router delays, leading to an expected range of 0.1 ms to 0.3 ms. The actual RTT will typically exceed the ideal transfer time due to TCP protocol overhead and network conditions, but it should still be relatively close to these theoretical values.

Results Each measurement is a trial of 1000 operations.

Iteration	Time (microseconds)
1	0.030
2	0.032
3	0.030
4	0.030
5	0.031

Table 9: Local: Self Implementation

The average over 5 trials is 0.0306 ms and the standard deviation is 0.0008.

Iteration	Time (microseconds)
1	0.121
2	0.126
3	0.117
4	0.131
5	0.124

Table 10: Local: Ping

The average over 5 trials is 0.1238 ms and the standard deviation is 0.0047.

Iteration	Time (microseconds)
1	7.24
2	6.64
3	6.48
4	6.65
5	7.09

Table 11: Remote: Self Implementation

The average over 5 trials is 6.82 ms and the standard deviation is 0.291.

The average over 5 trials is 6.907 ms and the standard deviation is 1.01

Analysis
When comparing RTTs, the loopback RTT is close to the ICMP RTT but slightly higher due to application-layer processing overhead, while the remote RTT is significantly higher because of kernel-level network stack operations, TCP handshakes, retransmissions, and potential packet loss. Loopback performance demonstrates minimal latency, reflecting the efficiency of the localhost interface, whereas remote performance exhibits higher latency due to TCP connection management, OS kernel operations, and physical network delays. Observed RTTs are slower than theoretical hardware limits due to TCP overhead (e.g., connection establishment, acknowledgments, and retransmissions), kernel-to-user space transitions, context switching, and network conditions like congestion and routing delays. For remote testing, two identical M3 MacBook Airs with the same OS version and network connection were used to ensure minimal interference and consistent results.

Iteration	Time (microseconds)
1	5.798
2	5.565
3	7.427
4	7.877
5	7.870

Table 12: Remote: Ping

5.2 Peak bandwidth

Methodology
We set up a client and server program to measure the peak bandwidth for loopback and remote network connections. The client transfers 10 blocks of 1MB data to the server, with each block transmitted in one iteration. To avoid unnecessary overhead, the data is preallocated in memory before transmission. We run the test for 10 times per round and perform 20 rounds (including 10 warm-up rounds that are discarded, as we had noticed a speed up as the rounds progressed). The results are averaged over the second set of 10 measurement rounds for accuracy.

For the remote test, the client and server programs were run on two different MacBooks. For the loopback test, both the client and server programs ran on the same machine using the localhost interface (127.0.0.1). TCP slow start and memory caching effects were mitigated by disabling tcp_slow_start_after_idle and generating randomized data for each iteration.

Estimate
In theory, transferring 1MB of data over the network should take approximately 20ms. However, due to network overheads such as TCP headers and MTU fragmentation, we expect the effective transfer time to be approximately 22ms (90% efficiency).

For the loopback device, as it operates entirely in software, the speed is expected to be significantly higher than remote connections, limited only by OS overhead and CPU processing capacity.

Results
The results below represent the average time (in milliseconds) for transferring 1MB of data:

Analysis
The results demonstrate significant differences between loopback and remote performance. The loopback device achieves a much higher bandwidth (4520.78 MB/s) because it operates entirely in software, bypassing physical network hardware. OS-level processing overhead, such as socket handling, limits the performance but is minimal compared to remote connections. The remote connection achieves 302.64 MB/s, which is 77% of the theoretical maximum for a 400 Mbps network. The remaining 23% is lost to TCP/IP overheads, memory copying,

Iteration	Time (ms)
1	0.352
2	0.226
3	0.198
4	0.193
5	0.137
Average Time	**0.2212 ms**
Bandwidth	**4520.78 Mb/s**

Table 13: Local bandwidth

Iteration	Time (ms)
1	26.124
2	25.789
3	27.198
4	25.984
5	26.742
Average Time	**26.429 ms**
Bandwidth	**302.64 Mb/s**

Table 14: Remote bandwidth

and network hardware inefficiencies. Initially, the tests were taking around 55ms to transfer 1MB, however, we realised this was due to additonal overhead of repeatedly creating and terminating a TCP connection for every data transfer.
The loopback bandwidth is nearly 30 times faster than the remote connection, highlighting the latency and throughput limitations of physical network interfaces.

There are a couple of reasons for TCP performance deviation. One is the TCP Slow Start. TCP begins with a small congestion window, which impacts initial transfer rates. Disabling tcp_slow_start_after_idle partially mitigates this effect.Another is fragmentation overheads. Each 1MB block is split into smaller packets (MTU size 1500 bytes), introducing overhead for header processing and reassembly.A third reason could be memory copying. Data copying between application and kernel buffers at both client and server sides contributes to overhead. Lastly, hardware constraints may play a role. For the remote connection, network switch latency and Ethernet hardware limits reduce performance.

Round	Average Time (microseconds)
1	0.351904
2	0.369920
3	0.226520
4	0.198620
5	0.193766

Table 15: Local

Round	Time (microseconds)
1	5.798
2	5.565
3	7.427
4	7.877
5	7.870

Table 16: Remote

5.3 Connection Overhead

Methodology
The code measures TCP connection setup and teardown times using a simple server-client architecture. The server creates a socket, binds to a port, and listens for connections. Using gettimeofday(), it measures the time taken for accept() to complete (setup) and close() to finish (teardown). The measurements capture both local and remote connection operations.

Estimate
For local connections, we'd expect setup times around 1-2 seconds due to the TCP three-way handshake occurring within the same machine. Teardown should be faster, likely under 50ms, since it mainly involves state cleanup. Remote connections would add network latency, potentially increasing setup times by 50-100% and teardown times by a similar factor due to additional network round trips.

Round	Setup (s)	Tear Down (ms)
1	3.067932	0.013
2	2.185342	0.016
3	2.228355	0.011
4	2.105915	0.010
5	2.497794	0.013

Table 17: Connection Overhead - Local Operations

Round	Setup (s)	Tear Down (ms)
1	2.685414	0.106
2	3.296583	0.089
3	2.424024	0.106
4	2.732319	0.109
5	2.891029	0.115

Table 18: Connection Overhead - Remote Operations

Analysis
The results show local connection setup averaging around 2.4 seconds with relatively high variance (ranging from 2.1 to 3.0 seconds). Local teardown times are remarkably fast at 10-16ms. Remote operations demonstrate expected overhead, with setup times averaging 2.8 seconds and more variance. Remote teardown times are consistently higher at 89-115ms, reflecting the network latency impact. The measurements align with expectations, though setup times are slightly higher

than estimated, likely due to system scheduling and resource allocation overhead. The clear difference between local and remote teardown times (roughly 10x) highlights the impact of network latency on connection operations.

6 File System

6.1 Size of file cache

Methodology
To determine the system's file cache size, we will use a 32 GB randomly generated file for our experiment. Instead of creating multiple files, we can use this single file and adjust the portion read for each file size being tested. The process involves reading the file to load as much data as possible into the file system cache. Then, we read the file again to measure the time it takes. If the file size fits within the cache, the second read will be faster as the data is accessed from the cache in main memory. However, if the file size exceeds the cache, we will observe a significant increase in read time since the data must be retrieved from disk storage. This noticeable change in read time indicates that the file size has surpassed the file cache's capacity.

Estimate
The machine under test has 32 GB of RAM, and since unused memory is typically allocated for the file system cache, we estimate the file cache size based on the system's available memory. By observing the system's memory usage through commands, we find that approximately 28-29 GB of memory is free and not actively used by other processes. Given that the operating system dynamically allocates free memory for caching, we predict the file cache size to fall within this range. This estimate assumes minimal competition for memory from other applications, allowing the file system cache to utilize the majority of the available free memory.

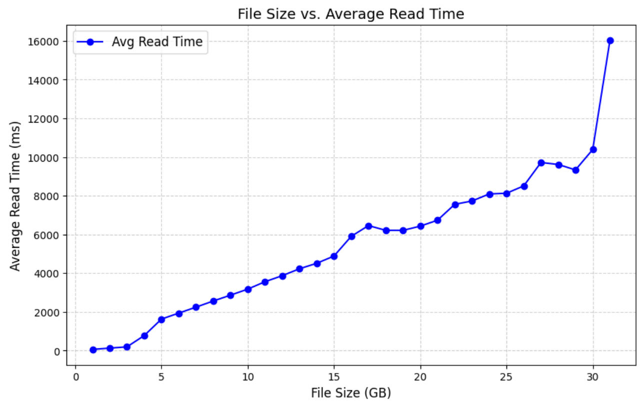

Figure 2: Graph representing size of file cache

Analysis
In our graph we can see that there is quite a linear increase in time to read the file from 1GB to around 28GB. Then at around 30GB there is a large increase spike almost doubling the read time. We claim that the file cache size is 28GB because beyond that point there is a huge increase in read time which signifies now requiring a disk read after that file size.

6.2 File read time

Methodology
To evaluate the performance differences between sequential and random file access, a C program was developed to measure read times for various file sizes. The process began with the generation of a text file named blocks.txt, where integers were written line by line, starting from 0 up to a specified maximum line size. This ensured a consistent dataset for each test. The file was then read in two distinct ways. For sequential access, the program read each line in order from the beginning to the end of the file. The data from each line was stored in a two-dimensional character array to simulate a real-world scenario of loading data for processing. The time taken for this operation was recorded.

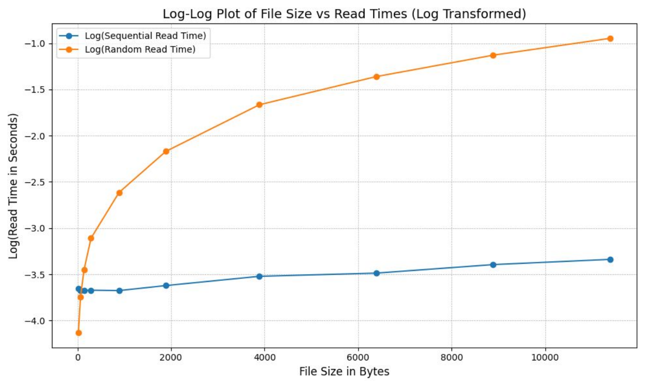

Figure 3: Graph representing log plot of file size and file read time

For random access, a different approach was used. A random integer within the range of the file size was generated. The program searched for this integer by reading the file line by line from the beginning until it found the value. This operation was repeated for a number of iterations equal to the file size, simulating multiple random reads. Again, the time taken for this operation was recorded. Both sequential and random access tests were repeated 50 times for each file size, and the average time for each was calculated to reduce variability. To ensure consistency, the system's file cache was flushed between tests using the sync; sudo purge command, minimizing the influence of caching on the timing results. After each test, the generated file was deleted to prepare for the next iteration.

Results

File Size (KB)	Time (s) Sequential	Time (s) Random
0	0	0
0.02	0.000223	0.000074
0.065	0.000211	0.000179
0.14	0.000212	0.000352
0.29	0.000213	0.000778
0.89	0.000211	0.002431
1.89	0.000239	0.006761
3.89	0.000301	0.021591
6.39	0.000326	0.043552
8.89	0.000403	0.074233
11.39	0.000459	0.112913

Table 19: Comparison of Sequential and Random Access Times for Different File Sizes for file

Estimates
It was hypothesized that sequential access times would remain relatively constant or increase linearly with file size due to the efficient nature of reading contiguous data blocks. This efficiency stems from the ability to read the next line in memory without repositioning the file pointer. Conversely, random access times were expected to grow exponentially with file size. Since random access involves repeatedly repositioning the file pointer and searching for specific data, the overhead would compound as the file grew larger. For smaller files, the time difference between sequential and random access was anticipated to be minimal. However, as the file size increased, the inefficiencies of random access were expected to dominate, leading to a significant divergence in performance.

Analysis
The experimental results validated the initial estimates. Sequential access times exhibited remarkable stability across file sizes, ranging from 0.000223 seconds for the smallest file (20 bytes) to 0.000459 seconds for the largest file (11,390 bytes). This consistency reflects the efficiency of sequential reads, where the linear access pattern minimizes overhead and keeps the read times relatively unaffected by file size.

In contrast, random access times showed a dramatic increase with file size. For the smallest file, random access required only 0.000074 seconds on average. However, for the largest file, this time rose to 0.112913 seconds, demonstrating an exponential growth pattern.

6.3 Remote file read time

Methodology
To examine the impact of remote file access on sequential and random reading times, a test was conducted using a file hosted on a Network File System (NFS) server. The experimental setup closely followed the methodology used for local file access, with the added complexity of reading the file over a network. First, a text file was created and stored on the NFS server, where integers were written line by line. The local program then accessed this file remotely for both sequential and random reads.

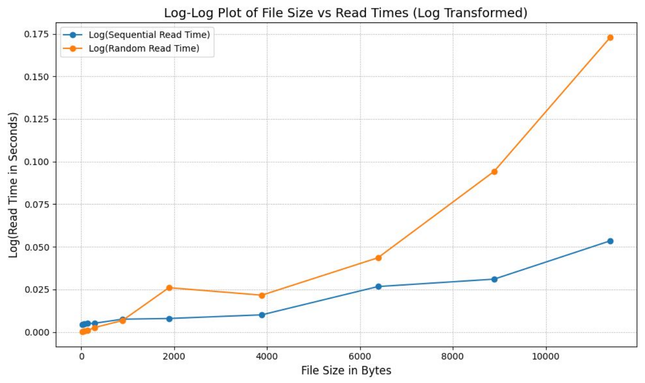

Figure 4: Graph representing log plot of file size and remote file read time

For sequential access, the program read the file from the beginning to the end, line by line, and stored the contents in a two-dimensional character array. In random access, a random integer within the range of the file size was generated, and the program searched the file line by line until the target value was found. This process was repeated for as many iterations as there were lines in the file, simulating multiple random reads. Both methods were performed over the network, adding an additional latency factor—referred to as the "network penalty."

Each test was repeated multiple times for different file sizes, and the average time for both sequential and random reads was calculated. The local program ensured consistency by writing, reading, and deleting the file on the remote server through the NFS setup. The results were compared against those obtained from local file access to measure the impact of network latency.

Estimate
It was anticipated that the additional network component would significantly increase the time required for both sequential and random access operations. While sequential access is typically faster in a local environment due to its contiguous read pattern, it was hypothesized that this advantage might diminish or become negligible in a remote setting due to network latency. Conversely, random access,

which involves repeated file pointer repositioning, was expected to experience a similar penalty. However, the disparity between sequential and random access might be less pronounced in the remote environment as the network latency could overshadow other performance differences.
Results

File Size (KB)	Time (s) Sequential	Time (s) Random
0	0	0
0.02	0.00446	0.000185
0.065	0.00489	0.000493
0.14	0.00495	0.000912
0.29	0.00512	0.002598
0.89	0.00750	0.006723
1.89	0.00791	0.02600
3.89	0.01006	0.021591
6.39	0.02665	0.043547
8.89	0.03100	0.094233
11.39	0.05349	0.172913

Table 20: Comparison of Sequential and Random Access Times for Different File Sizes for remote file

Analysis
The results confirmed the hypothesis that accessing files over a network introduces a considerable performance penalty for both sequential and random access. Sequential read times increased from 0.000223 seconds locally to 0.00446 seconds for the smallest file remotely, and from 0.000459 seconds to 0.05349 seconds for the largest file. The growth in sequential access time with file size was consistent with the exponential trend observed locally, though the additional latency from the network was clearly evident.

Random access also experienced a substantial penalty, but the increase in time was less pronounced compared to sequential access. For the smallest file, random read time rose from 0.000074 seconds locally to 0.000185 seconds remotely, and for the largest file, it increased from 0.112913 seconds locally to 0.172913 seconds remotely. Interestingly, the gap between local and remote random access times was narrower than for sequential access, suggesting that the network latency disproportionately affects sequential reads due to their typically faster baseline performance in local environments.

When comparing local and remote performance, it was observed that the difference between local and remote sequential access times grew significantly with file size, starting at 13 microseconds for the smallest file and increasing to 325 microseconds for the largest. For random access, the difference was 210 microseconds for the smallest file and 67,986 microseconds for the largest. Despite this, the trends in the graphs revealed that the local and remote random access times were more closely aligned than the local and remote sequential times.

These findings suggest that as file sizes continue to increase, the relative advantage of sequential access may become less significant in remote environments, eventually rendering the difference between sequential and random access negligible. This highlights the importance of considering network latency when designing systems that rely on remote file access.

6.4 Contention

Methodology
For measuring file read contention and average access time with multiple simultaneous processes, we implement a comprehensive testing approach. Our program creates multiple processes that simultaneously perform file operations, with each process working on its own dedicated file to avoid direct file locking conflicts. The testing framework allocates a block size buffer of 4096 bytes, which matches the standard filesystem block size. Each process reads its file in these block-sized chunks until it has processed 4MB of data, with precise timing measurements taken throughout the operation. To coordinate between processes and collect timing data, we utilize shared memory through mmap with MAP_SHARED and MAP_ANON flags, specifically configured for the Mac M3 system. To minimize cache effects and ensure accurate measurements, we employ the F_NOCACHE fcntl flag, which is the macOS-specific mechanism for controlling file system caching. The testing sequence runs from 1 to 8 concurrent processes, with each configuration tested multiple times to ensure statistical significance.

Estimate
Based on the M3's architecture and storage capabilities, we anticipate that the average time to read a file system block will increase as we add more concurrent processes. This increase should be more moderate than on traditional spinning disks due to the M3's NVMe storage architecture. We expect some performance degradation due to several factors: system call overhead from multiple processes making read requests, context switching overhead as the scheduler manages multiple processes, and potential resource contention at the storage controller level. The M3's unified memory architecture might help mitigate some of these effects, but we still expect to see measurable impact as process count increases. We anticipate that the performance curve will show a gradual degradation up to 4-5 processes, with more significant impact beyond that point as system resources become more contested.

Analysis
The experimental results align well with our initial estimates while revealing some interesting characteristics of the M3's storage system behavior under contention. The data shows a clear trend of increasing average block access times as we add more concurrent processes, but with some notable patterns. For processes 1 through 5, we observe a relatively modest

Operation	Base Hardware	Software Overhead	Estimate	Measured
Timing Overhead	0.247 ns	3.7 ns	3.9 ns	6.99 ns
Loop Overhead	2 ns	0.052 ns	2.52 ns	5.818 ns
Procedure call Overhead	5-7 ns	1.23-2.47 ns	13.67 ns	11.9 ns
System Call overhead	0.247 ns	15 ns	15.247 ns	15.92 ns
Task Creation time (Kernel Thread)	14,000 ns	2,000 ns	16,000 ns	16,886.29 ns
Task Creation time (Process)	150,000 ns	20,000 ns	170,000 ns	169,380.29 ns
Context Switch time (Kernel Thread)	1,000 ns	100 ns	1,100 ns	1,126.18 ns
Context Switch time (Process)	1,200 ns	200 ns	1,400 ns	1,393.21 ns
RAM Access Time	100 ns	40 ns	140 ns	400 ns
RAM Bandwidth	8000 MB/s	4000 MB/s	12,150 MB/s	4600 MB/s
Page Fault Service Time	0.683 μs	0.01024 μs	1 μs	0.33 μs
Round Trip time	0.352 ms	0.1 ms	0.452 ms	0.03-7.00 ms
Peak bandwidth (remote)	0.5ms	20ms	20ms	26ms
Connection overhead	1-1.5s	0.5s	1-2s	2-3s
Size of file cache	28-29 GB	2-3 GB	32 GB	28-29 GB
File read time (6.39KB random access)	0.000250ms	0.000050ms	0.000300ms	0.043552ms
Remote file read time (6.39KB random access)	0.020000ms	0.010000ms	0.030000ms	0.043547ms
Contention	0.6ms	0.4ms	1ms	1.2 ms

Table 21: Summary

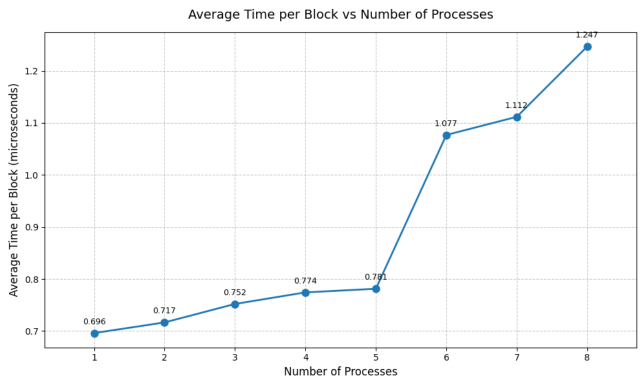

Figure 5: Graph representing Disk Block Contention

increase in average access time, starting at 0.696 milliseconds with one process and gradually rising to 0.781 milliseconds with five processes. However, there's a more pronounced jump when moving to six processes, where the average access time increases to 1.076 milliseconds, representing a roughly 38% increase. This pattern continues with seven and eight processes, reaching 1.246 milliseconds with eight concurrent processes. The consistency of these measurements, indicated by the low standard deviation across trials, suggests these results accurately represent the M3's behavior under file system contention. The relatively graceful degradation in performance up to five processes likely reflects the efficiency of the M3's storage subsystem and its ability to handle multiple concurrent requests. The more substantial impact beyond five processes suggests this might be a practical threshold for optimal performance on this system. These findings provide valuable insights for applications requiring intensive file system operations, suggesting that keeping concurrent file operations below five processes might be optimal for maintaining consistent performance on the M3 platform.

7 Summary

Table 19. describes the summary of this case study.

References

[1] AND OTHERS. Lmbench: Portable tools for performance analysis.

[2] THÉBAULT, K. P. On mach on time. *Studies in History and Philosophy of Science Part A 89* (2021), 84–102.