Justin Tahara, Shaya Parsa, Imran Matin

Professor Voelker

CSE 221

30 January 2021

CSE 221 System Measurement Project

Table of Contents:

Introduction:

Our goal for this project was to improve our understanding of an operating system that we use on a daily basis. We wanted to learn how to effectively benchmark and test the performance of large, complex systems. MacOS is a widely used operating system however, we believe that most users of the operating system don’t have a proper understanding of how it actually works. Our team chose to use Imran’s 2017 Macbook Pro as the main device for experimentation. Justin and Shaya also contributed by conducting research and providing the necessary code in order to carry out the experiments. All experiments were performed as a group multiple times to ensure reliability in our results and understanding across the whole group. We implemented all of our experiments with C. The C compiler we used is GCC and the version is Apple clang version 12.0.0 (clang-1200.0.32.29). The flags that we compiled our code with the C compiler are -lm -lpthread -O0 and we used a *Makefile* to automate compilation and facilitate testing. Since we were using a standard 2017 MacBook Pro with large amounts of storage and RAM, we didn’t expect any major implications from the environment. However, we did expect to run into issues with noise and variance in our performance measurements since our system may have other processes running simultaneously as well as having older hardware. We also expected to run into issues when performing the tests because MacOS is a commercial OS and not open source, therefore we may not have complete access to the underlying system. We estimated 50 hours, but after completion of the project, we believe we spent a total of 120+ hours.

Machine Description:

We found a lot of information about our specific machine from this link.
Machine: MacBook Pro (13-inch, 2017)
Processor Model: Dual-Core Intel Core i7 (I7-7660U)
Cycle Time: 2.5 GHz
Cache Sizes:

- L1 Instruction (per core) = 32KB
- L1 Data (per core) = 32KB
- L2 (per core) = 256KB
- L3 = 4MB

DRAM Type: LPDDR3
Clock: 2133 MHz
Capacity: 8GB (one slot), 16GB (total)
Memory bus bandwidth: 34.128 GB/s
I/O bus type: 2x Thunderbolt ports, 3.5 mm Audio Port, assuming PCIe 3.0
Bandwidth: Thunderbolt Speed = Up to 40 Gb/s, USB 3.1 Gen 2 (up to 10Gb/s)
Disk (SSD):

- Model Name: Apple SSD AP0256J
- Capacity: 250.69GB
- Transfer Rates: Unknown
- IOPs: Unknown
- Latencies: Unknown
- Note that since this SSD is an Apple commercial disk, we were having trouble finding information about this.

Disk (Hard Drive): N/A
Network card bandwidth: 1300 Mbps
Operating system (including version/release): macOS Big Sur, Version 11.1

Testing Environment Setup

The Operating System that we are using on the ieng6 server is CentOS Linux 7. Below are the system specifications of the ieng6 server.

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Number of CPU(s):	8
Thread(s) per core:	1
Core(s) per socket:	1
Socket(s):	8
CPU Model:	Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	20480K
Page Size:	4096K
OS Name:	CentOS Linux
OS Version:	7 (Core)
RAM Size:	62G (whole server)

The Operating System that we are using on the Amazon EC2 instance is Ubuntu 180.04LTS Linux. Below are the system specifications for the EC2 instance.

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Number of CPU(s):	1
Thread(s) per core:	1
Core(s) per socket:	1
Socket(s):	1
CPU Model:	Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	30720K
Page Size:	4096K
OS Name:	Ubuntu
OS Version:	18.04.5 LTS (Bionic Beaver)
RAM Size:	30GiB (32GB)

CPU, Scheduling, and OS Services

a. Measurement Overhead:

i. Time Counter Read Overhead

Methodology:

To measure the time taken for code execution, our group decided to use the function *RDTSC* from the *x86intrin.h* library. Our motivation for this choice was derived from the white paper How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures by Gabriele Paoloni. *RDTSC* provides a way to measure clock cycles, which provides us with very fine grained time data. The code provided in Paoloni's paper also handles the problems of register overwriting and out of order instruction execution (*CPUID* and *RDTSC/RDTSCP*) as well as minimizes the overhead in taking time measurements. We knew the machine's CPU clock cycle is 2.5 GHz, therefore we were able to calculate the conversion factors for cycles to seconds, microseconds, and nanoseconds.

Therefore, to calculate the overhead for reading the time counter, we just measured the time difference between two adjacent calls to RDTSC in a for loop. We executed a total of 1000 samples to have enough data to find the true overhead in reading time.

We convert cycles to nanoseconds according to the following equation:

$$2.5\ GHz\ =\ \frac{2.5 \times 10^{9}\ Cycles}{1\ sec} \times \frac{1\ sec}{10^{9} nanoseconds} = 2.5 cycles\ /\ ns$$

Using the above conversion factors for cycles to time as well as different statistical measures we were able to gather our final overhead measurements for reading the time counter.

Prediction:

Base Hardware: Since this experiment is software oriented, we don't expect there to be any base hardware overhead.

Software Overhead: We researched articles online in order to get a gauge of the measurement overhead for reading time with the *RDTSC* function. Our initial prediction is based on assuming the RDTSC asm executes 6 instructions and therefore 6 cycles for a time of 15 ns. In another article here, we read that it could take 20-25 cycles for a time of 62.5 ns.

Results:

Average Time Taken (ns)	Median Time Taken (ns)	Std Dev. (ns)
12.032	10.800	2.027

Analysis:
Reviewing our results, we see that the both the average and median time taken measurements are very similar, and that those measurements are slightly faster than the predictions made in the article referenced in the previous paragraph. The variability is also low in this measurement as the standard deviation is small. We attribute this to the strong hardware that we have on the machine we are using.

ii. Loop Overhead

Methodology:
We conducted the loop overhead experiment by using two loops. The outer loop was used to execute the experiment 1000 times and therefore gather 1000 samples. The inner loop is the minimal loop that contains just a *continue* statement so as to contain no instructions. Thus, the overhead we are measuring in this experiment is the overhead for this minimal loop.

We decided to execute the minimal loop 1000 times (number of samples taken) because we assumed that most of our experiments that have a loop would utilize that many samples. We measured the time of the inner loop by reading the time before the inner loop, and after the inner for loop. We then took the average of the 1000 samples gathered and recorded the average time taken.

Prediction:
Base Hardware: Since this experiment is software oriented, we don't expect there to be any base hardware overhead.

Software Overhead: Since we execute the loop 1000 times, the actual time taken per iteration of the loop should be magnified by 1000. Following this article and this link to the website godbolt, we can look at the assembly line code for the for loop operation to estimate the cycle count which seems to be roughly 5 ~ 6 cycles. Using the cycles to nanoseconds conversion from part i, we see that 6 cycles per loop iteration times 1000 loop iterations gives us 6000 cycles total for a total time of 1500 ns. We acknowledge that this is a rough estimate since we do not take into account branch prediction errors and pipelining when converting instructions executed per cycle to a time measurement.

Results:

Average Time Taken for 1000 iter. (ns)	Median Time Taken for 1000 iter. (ns)	Std Dev. for 1000 iter. (ns)
1983.2096	1878.0000	3221.4096

Analysis:

We see from our experiment that the average time taken was roughly 2000 nanoseconds. Therefore, using the assembly line code calculations from the previous sentences, we see that our predictions and results are valid. The disparity in the times as discussed in our predictions is that we are not considering many optimizations that compilers and modern hardware perform on minimal loops. This also explains why there is such a large standard deviation between our samples. So it is reasonable that the observed time is very similar to what we expected.

b. Procedure Call Overhead:

Methodology:

To measure the overhead of procedure calls, we created 8 different functions and each function was designed to handle a different number of arguments. The motivation behind having multiple functions with different numbers of arguments was to record the differences in overhead due to the number of arguments. In order to focus on the overhead due to the number of arguments rather than the contents of the function, we ensured the functions themselves were minimal. A minimal function is one that only contains a return statement and that is it. We conducted the experiment 1000 times per function in order to gain enough samples, and then recorded the average time taken of those samples.

Prediction:

Base Hardware: Since this experiment is software oriented, we don't expect there to be any base hardware overhead.

Software Overhead: We would expect that calling the functions with different amounts of arguments would cause a series of *MOV* operations per argument, in order to move the argument values into the registers. Considering this fact and according to our research using this article, a typical *MOV* instruction can take anywhere from (theoretically) 1 cycle to order of 100 cycles. If we predict that on average each *MOV* instruction takes 100 cycles, then we would expect the overhead to be roughly 125 nanoseconds for 50 cycles multiplied with the number of arguments the function has.

Result:

Number of Arguments	Average Time Taken (ns)	Median Time Taken (ns)	Std Dev. (ns)
0	12.425120	10.800000	2.699730
1	19.939520	12.000000	3.301204
2	12.000000	12.353360	7.516833
3	18.144160	12.000000	8.057899
4	15.401760	14.400000	8.885459
5	12.933760	12.800000	7.157081
6	12.738640	12.000000	21.476651
7	17.823760	14.800000	5.454669

Analysis:
We assumed that if there were more arguments it would take longer to call the corresponding function but there didn't seem to be a correlation between having multiple arguments compared to having no arguments at all. Our predictions could have been wrong because we assumed a large number of cycles for the *MOV* operation.

Another factor in the execution of this code is that instructions are executed in a variable amount of cycles, therefore the total cost could have been amortized across the whole code execution. Since we are also using a computer with a modern CPU, 16GB RAM, the execution could also be magnitudes faster, which would explain the results we have gotten. It is important to note that modern machines use aggressive pipeline and parallelism to execute instructions, therefore the overhead of the *MOV* instructions could be minimal due to hardware optimizations. These hardware optimizations we believe are also the reason for the large standard deviations in our measurements. However, the number of samples we acquired should allow us to handle this variability.

From our experiment with testing the overhead of RDTSC, we see the overhead in this experiment are very similar. We are interpreting this as procedure calls are very efficient on this machine since the main overhead is coming from our benchmarking code. Therefore, we see the increment overhead of an argument in our results is so minimal that it cannot be seen.

c. System Call Overhead:

Methodology:
To perform this experiment we measured two syscalls: *getpid* and *kill*. For each system call, we gathered 1000 samples by executing each syscall in a C for loop and recording the time taken.

For the *getpid* syscall, a minimal syscall, we simply get the current time, execute *getpid*, and then get the end time. We have two versions of this test. The first runs a C for loop for 1000 iterations to gather samples, whereas the second only runs *getpid* once but executes this experiment 1000 times using a bash for loop. The reasoning for this is because we were seeing very different results from the two methods of sampling, therefore we thought it was best to analyze both.

For *kill* we had to do a little bit of setup by forking a new process that we could kill. If the child process executes first, the child process will execute the *raise* syscall with the *SIGSTOP* flag, therefore context switching to the parent process. The parent process will then take the current time, execute the *kill* syscall with the *SIGKILL* flag, and then take the end time. We then reported the average overhead time per system call. Note, to gather our results, we only executed one syscall test at a time to avoid noise in our results.

Prediction:
Base Hardware: Since this experiment is software oriented, we don't expect there to be any base hardware overhead.

Software Overhead: For the *kill* syscall, we could not find reliable benchmarks so we will reason out a prediction. The functionality of this syscall is to send a signal from one process to another. Therefore, since we are sending a signal to another process we will be using interprocess communication (IPC). There is overhead with this (in our case this is a procedure oriented system), as the kernel handles this. Therefore, we predict the overhead of this syscall to be ~500ns.

We believe that the *getpid* syscall is very minimal, therefore it should really only have the overhead of crossing the protection boundary into the kernel and back. Our procedure call overhead was ~15 ns in the Procedure Call Overhead Experiment, therefore our predictions for the overhead of syscalls should be much larger due to the costly operation of crossing a protection boundary into the kernel. We expect to see a total overhead time of ~200 ns for this syscall.

Results:

System Call	Average Time Taken (ns)	Median Time Taken (ns)	Std Dev. (ns)
getpid() (C loop)	20.140000	12.000000	318.794382
getpid() (bash loop)	14080.99639	11831.2	6697.515962
kill()	4395.408800	3658.800000	5334.891857

Analysis:

We will start by analyzing the results for *getpid*. We find it very interesting that the two different sampling methods had such different results. It is tough to compare these results against the *kill* syscall results because the functionality is much different, therefore we will reason about these results. It could be that when running the experiment in the bash loop, we are continuously spawning new processes which are altering the OS' data structure for tracking process ids very rapidly. Therefore, the operation is taking much longer because there could be locking on this data structure to ensure that it is consistent when processes read it, and therefore causing processes to wait for this. In the case of the C loop experiment, we are only experiencing the overhead for the loop and the OS' process id data structure is not being changed during the test. Therefore, reading this data structure for the process' id is much faster. Another point that must be made is that the system is potentially caching the *getpid* syscall when it is run in the C loop, therefore calling it multiple times sequentially is not truly calling the syscall but instead going to the cache which is much more efficient.

The results for *kill* are less surprising when compared to *getpid*. We believe that the *kill* syscall has much more overhead because it must use interprocess communication, in this case procedure calls into the kernel, to complete its functionality. *Kill* will also return a value for whether the signal sent was a success, therefore it may wait to see if the signal it sent was a success which could take much longer.

For both system calls, we can see that the overhead is much larger than for a procedure call. Yes, the *getpid* experiment has varied results, but the *kill* syscall is clearly much more costly. As discussed earlier, this difference in overhead is due to crossing the protection domain into the kernel to execute the syscalls. As discussed in previous sections, the large variability in all of our measurements most likely come from the optimizations in modern hardware.

d. Task Creation Time:

i. Thread Creation Overhead:

Methodology:

We measured the thread creation overhead by first reading the current time, then creating a new thread with the *pthread_create()* function from the *pthread.h* library. *pthread_create()* creates a new thread and will both execute the function specified when it is run and return the thread id for that thread to its parent. We will not know whether the parent or the child or another process or thread will run immediately after the thread creation, therefore we placed two different statements to read the end time for thread creation, one at the beginning of the child thread and one right after the call to *pthread_create()* in the parent. Unfortunately, we are unable to know whether the parent or the child is the next thread context switched to, therefore there could be excess time measured due to the context switches. We handle this by using an upper bound 20,000 ns for the time taken to create a thread, so if a measurement is larger than this we repeat this sample (refer to the analysis section for the reasoning behind this upper bound). We also ensure that the parent waits for the child thread to complete by using *join* after it has taken its end time. The reason for this is because we may switch from the parent to the child and back to the parent. Then, if we do not join we will get the wrong end time value since the child has not returned it's clock cycle value even though it is the most accurate one. Once we have both end times, we compare to see which one has a lower cycle count in order to get the most accurate overhead of thread creation.

We then run this experiment 1000 times using a bash script to record all of our results and calculate the average and median times for thread creation.

Prediction:

Base Hardware: Since this experiment is software oriented, we don't expect there to be any base hardware overhead.

Software Overhead: Looking at this article, we see that there is a comment made by a user stating that their machine takes roughly 10 microseconds in order to create a thread. Their machine is using an *Intel Core i5-2540M* which is from 2011 which would be 10 years ago. We know that the machine used may be outdated, but the process of experimentation is the same so with some hardware improvements, we would assume our results will be slightly faster for a predicted overhead of ~10,000 ns. We expect the overhead to be small compared to the overhead for process creation because we know that processes are lightweight abstractions.

Results:

Average Time Taken (ns)	Median Time Taken (ns)	Std Dev. (ns)
11754.554399999999	10910.9668076800	3189.126610825879

Analysis:
From our results, we can see that the time taken to create a thread is relatively the same compared to the article that we found. 10 microseconds converted to nanoseconds would be 10,000 nanoseconds meaning our times are basically the same. We see from our results for process creation that it is much faster to create threads rather than processes, which we attribute to the fact that threads are much more lightweight abstractions. The slight difference can be attributed to random noise when context switching between different threads or processes. If there are many of these random context switches in our samples, our data can be skewed.

We attempted to limit the noise in our results from context switching by upper bounding our measurements at 20,000 ns since those can be considered invalid for measuring the cost of thread creation. The upper bound allows us to remove the outliers we were initially seeing in our results, and these outliers are occurring because they measure the time it takes to create the thread as well as many context switches between threads and processes before we measure the end time in either the parent or child thread. We only want to gather times for thread creation, therefore this allows us to filter out those times that include extra operations. There still exists variability in our experiment, however the standard deviation is acceptable as it is a complete order of magnitude smaller than our measurements. There will always be noise in this experiment since we cannot control context switches.

It is important to note that we used the bash script method to test because we were initially getting results that showed the time for the thread creation increased as we gathered more samples in the same program. We understood this as invalid and that is why we changed our methodology.

We were however, surprised that the article experimented with 2011 hardware and we are using modern hardware, yet, our times are very similar. We think this is due to the fact that in order to create a thread, the operations that need to happen are still the same and so the hardware difference might not play a key role in optimizing thread creations.

ii. Process Creation Overhead:

Methodology:

The process creation overhead experiment was performed as follows. We first create a shared pipe for data transfer between the parent and child process. Next we take the current time as our start time. We then execute the *fork* system call to create a new child process. Once we have created the child process, there are 2 cases that occur since we cannot determine whether the parent or the child process will execute first.

Case 1: Parent Process Executes First
In this case, the parent process will execute first and will take the current time as the end time. The parent process will then continue executing and execute the *waitpid* and *kill* syscalls to ensure the child process terminates before continuing.

Case 2: Child Process Executes First
In this case, the child process will execute first and will take the current time as the end time. It will then write the end time into the shared pipe for the parent process to read and exit.

At this point, the parent process will be executing because either the child process terminated or the parent process waited until the child process terminated. Next the parent process will read from the shared pipe to retrieve the end time taken from the child process. Since we do not know whether the child process or parent process executed first after the *fork*, we compare the end time taken in the parent and the end time taken in the child and use the one that is earlier. The reason for this is because we only want to measure the process creation time, therefore we want the time that was taken as soon as possible after the call to *fork*. Finally, we compare the time taken to an upper bound of 1,000,000 ns (the reasoning behind this upper bound is discussed in the analysis section). If the time is below the upper bound then we save it, else we don’t.

We then run this experiment 1000 times using a bash script to record all of our results and calculate the average and median times for process creation.

Prediction:

Base Hardware: Since this experiment is software oriented, we don't expect there to be any base hardware overhead.

Software Overhead: According to Table 9 of the lmbench paper, we see that process creation time should be roughly 0.5 ~ 1 milliseconds. However, the machine that was used to get these results was extremely old and so we predict that with our machine's hardware we should see a process creation overhead that is faster than 0.5 ~ 1 milliseconds.

We also know that processes must initialize much more data than threads during creation because they must create large data structures like the Process Control Block (PCB). This overhead does not exist for creating threads, therefore we expect this to increase the overhead of process creation by a large amount of time.

Results:

Average Time Taken (ns)	Median Time Taken (ns)	Std Dev. (ns)
174440.24979253113	149926.0	132257.7841441311

Analysis:

We can see in our results that the average time taken to create a process is 174440 ns whereas the average time taken to create a thread is 11754 ns. These times show that it takes a complete order of magnitude longer to create a process than to create a thread. We expected this difference as processes are much heavier weight abstractions than threads. As we discussed in our prediction, processes must initialize large data structures like the PCB, and therefore have a lot more overhead than threads during creation.

We overpredicted the overhead for process creation, and we assume that this is because we based our prediction off of hardware that was much older than our machines. Modern hardware today is extremely optimized, and therefore this will greatly speed up the operation of process creation.

We also determined that experiments that had times greater than 1000000 ns were invalid for measuring the cost of process creation The upper bound allows us to remove the outliers we were initially seeing in our results, and these outliers are occurring because they measure the time it takes to create the process as well as many context switches between threads and processes before we measure the end time in either the parent or child process. We only want to gather times for process creation, therefore this allows us to filter out those times that include extra operations. The variability in our data, as shown by the standard deviation, is large in this case. We attribute this to the fact that creating a process is a large operation that may take longer depending on the state of the machine as well as the fact that we cannot control context switches during process creation.

It is important to note that we used the bash script method to test because we were initially getting results that showed the time for the process creation increased as we gathered more samples in the same program. We understood this as invalid and that is why we changed our methodology.

e. Context Switch Time:

i. Thread Context Switch Overhead:

Methodology:

We first create a child thread using the *pthread_create()* function and specify that the created thread should run a function that will return the end time. Next, in the parent thread, we then read the current time. From here the parent thread will call the *pthread_join()* function in order to force a context switch to occur. Once we begin executing in the child thread, we read the current time and return that time to the parent thread (since it executed *join*) in order to calculate the time taken for a context switch. The difference between this experiment and the experiment in Part *d* is the location in which we read the start time (after the creation).

Note that due to the unpredictability of context switching, we cannot ensure that the child thread is immediately context switched to, and therefore there may be unwanted variability in our measurements. We handle this in two ways. The first is by wrapping all the code for the paragraph above in a while loop that has a condition checking that the difference between the start and end time (technically the number of cycles passed from start to end) is positive. If it is not positive, the test is run again. This is to handle the case of the child thread executing and returning before the parent thread has saved the start time. The second is by checking if the time we are recording is lower than the upper bound of 10,000 ns (the reason for this upper bound is explained below). This is to remove outliers (which we saw when analyzing our results) which occur because we do not directly context switch from the parent thread to the child thread.

We then execute this test using a bash script that will collect 1000 samples, and we use these to compute the average and median thread context switch overhead.

Prediction:

Base Hardware: Since this experiment is software oriented, we don't expect there to be any base hardware overhead.

Software Overhead: According to this article we see that generally context switches can take anywhere from 100 ~ 3000 nanoseconds, depending on the processor and the size of the context (in this case the size of the thread) that is to be saved and restored. We predict that our thread context switch time will be closer to 100 ns since our processor is very powerful and the overhead of saving and restoring threads is very low. This measurement should be a lot lower, thousands of ns lower, than the time required to context switch between processes.

Results:

Average Time Taken (ns)	Median Time Taken (ns)	Std Dev. (ns)
6996.019	7159.6	1539.2243498281855

Analysis:

We can see in our results the average and median times for our experiment were both ~7000 ns. This is significantly larger than what we predicted. We believe that one of the reasons for this is we estimated the weight of a thread to be lower than it actually is. Another reason for this misprediction is that we assumed our processor, though strong, was not as strong as we believe and therefore needed more time to perform this context switch. However, the overhead to context switch for a thread is much lower than the overhead to context switch for a process as we discuss in the next experiment.

The reason that we bounded our times from our samples as less than 10,000 ns is because we saw that we were not directly timing the one context switch between the parent thread to the child thread. Instead, we were measuring potentially many context switches both between processes and threads before we finally arrived at the child thread. For that reason, we bounded our valid times to ensure we were measuring only the time for one thread context switch. For measuring thread context switches, we are not worried about the standard deviation in our data because when compared to process context switches the overhead is still relatively small.

It is important to note that we used the bash script method to test because we were initially getting results that showed the time for the context switch increased as we gathered more samples in the same program. We understood this as invalid and that is why we changed our methodology.

ii. **Process Context Switch Overhead:**

Methodology:

The process context switch overhead experiment was performed as follows. We first create a shared pipe for data transfer between the parent and child process. We then execute the *fork* system call to create a new child process. Once we have created the child process, there are 2 cases that occur since we cannot determine whether the parent or the child process will execute first. If the child process executes *raise* first, the parent process will be context switched to and when the parent process executes *waitpid*, nothing will happen (we confirmed this when testing with print statements tracking to track execution). If the parent process executes *waitpid* first, the child process will be context switched to and when the child process executes *raise*, it will force a context switch back to the parent process (we confirmed this when testing with print statements tracking to track execution). At this point, we know we will always be in the parent process irrespective of which process executes first. Now, there are two cases that will occur for actually timing a context switch between two processes. These cases arise because we are unable to predict which process will execute first.

Case 1: Before Kill...After Raise...After Kill...

In this case, the parent process will begin by taking the start time. Next, the parent process will execute the *kill* syscall which forces a context switch to the child process. The child process will then begin executing after the *raise* syscall and read the end time. Finally, the child process will write the end time to the shared pipe and exit. This will force a context switch back to the parent process in which it will then read the end time from the child process.

Case 2: Before Kill...After Kill...Before Read...After Raise.

In this case, the parent process will begin by taking the start time. Then the parent process executes the *kill* syscall, and therefore we expect to see a context switch back to the child process. However, unlike what we assumed, it does not force a context switch to the child process and instead continues executing in the parent process (we elaborate more on why this situation occurs below). The parent process then executes the *read* syscall on the shared pipe, however the child process has not written the end time to the pipe yet. This forces a context switch back to the child process since this is a blocking pipe, and the child process will then take the end time, execute the *write* syscall on the shared buffer with the end time as its data, and then exit. This forces a context switch back to the parent process, and the parent process can now successfully execute the *read* syscall on the shared pipe to access the end time taken in the child process. Note however, that in this case the timing measurement includes the extra overhead of the parent process executing the *kill* syscall on the child process and shared pipe respectively.

Once we have the start and end times, we calculate the total time taken in nanoseconds. We then only select times that are below the upper bound of 1,00,000 ns. We explain why this is the upper bound selected in our analysis. We then run this experiment 1000 times using a bash script to record all of our results and calculate the average and median times for a process context switch.

We are unsure why the *kill* syscall does not consistently force a context switch from the parent to the child. However, after doing some research we believe that it could be that calling *raise* with *SIGSTOP* makes the child stop itself and context switches to the parent, but the child does not finish stopping itself before the parent executes *kill*. If that happens the *kill* call is void and the parent process continues to read, which resumes the child process. We acknowledge that there could be other reasons for why this is occurring that we have not understood.

Prediction:
Base Hardware: Since this experiment is software oriented, we don't expect there to be any base hardware overhead.

Software Overhead: According to the lmbench paper, they estimate that context switches between processes should be in the order of 20 microseconds. However, since we have modern hardware and a relatively strong machine, we would estimate a lower cost for process context switches. We think that the process context switch overhead should be roughly around 5000 nanoseconds due to our powerful processor. This measurement should be significantly higher, thousands of ns higher, than the time to context switch between threads because processes are much more heavyweight (PCB is very large on modern OSes).

Results:

Average Time Taken (ns)	Median Time Taken (ns)	Std Dev. (ns)
25929.59744680851	21828.4	10960.392370544414

Analysis:

We can see in our results that the median time taken to context switch between threads is 7159.6 ns whereas the median time taken to context switch between processes is 21828.4 ns. These times show that it takes a complete order of magnitude longer to context switch between processes. We expected this difference as processes are much heavier weight abstractions than threads. However, we underpredicted the time for a context switch between processes. We overestimated the strength of our processor, but more importantly this shows that we underestimated the size of processes. From previous courses, we know that Process Control Blocks (PCB) have continued to grow over time, and today they are very large. Therefore, saving and restoring PCBs are a very heavyweight operation and will take a lot of time.

As mentioned above, we have 2 cases and we can assume that each case happens with roughly a 50% chance. The second case has extra overhead of performing a *kill* system call. We measured the extra overhead of this system call in a seperate test (System Call Overhead) and it has a median time of *4395.4* ns. We did not subtract this amount from our measurements in the results section above, but we wanted to discuss the potential difference in our results caused by the *kill* system call.

We also determined that experiments that had times greater than 1,00,000 ns were invalid for measuring the cost of one context switch between processes. The upper bound allows us to remove the outliers we were initially seeing in our results, and these outliers are occurring because they measure many context switches between threads and processes before we measure the child process' end time. We only want to gather times for context switching directly from the parent process to the child process. This experiment still has a large standard deviation, but this is normal since the machine will try to optimize performing a large operation such as a process context switch and this is not always possible. Therefore, we will see noise in our data due to the unpredictability of when context switches are handled.

It is important to note that we used the bash script method to test because we were initially getting results that showed the time for the context switch increased as we gathered more samples in the same program. We understood this as invalid and that is why we changed our methodology.

Memory

a. RAM Access Time:

Methodology:

We began by first reading and understanding the paper lmbench: Portable Tools for Performance Analysis by McVoy and Staelin. We understand that most machines today use caching to optimize memory accesses and that most memory access patterns exhibit temporal or spatial locality. Therefore, we took the following approach.

In our experiment, for each size in the range 32KB ~ 1.5GB, we malloc an array of *chars* that we then loop through using different stride lengths ranging from 128 bytes to 16MB because we want to observe the effects of reading data from the different levels in the cache as well as this is what the lmbench paper used as well. We decided to use the different file sizes starting from lower than the size of the L1 cache all the way to larger than the size of the L3 cache (into main memory). For each combination of array size and stride length, once we have malloced the array, we fill the array with random *chars* in order to make sure the array is stored in memory. We then *malloc* another 16MB array that is used for the sole purpose of clearing out the cache of any data that we brought in while writing random characters to the array that we will be looping through. The reason why we initialize this flushing array to 16MB is because we want to make sure that data in cache is cleared properly. Looking at the size of the L3 cache on our machine which is 4MB, we decided to multiply that value by 4 to ensure that most or all of the data will be cleared.

Once we have completed the initialization process for the array we are going to read and we have completed flushing the cache, we enter our for loop where we begin our reads. The for loop is iterating 1000 times to amortize the time it takes to perform a read across each stride size, file size combination. Remember that the goal of this experiment is to see how long it takes for reading different file sizes with different stride lengths, therefore we want to have a statistically strong measurement and that is accomplished with many samples. Each time we iterate through the loop we first time the time it takes to access part of the array at some index. Initially that index is the length of the array that we malloced minus 1. We then subtract the stride length from the current index as we iterate through the loop in order to observe the effects of accessing different parts of the cache without the effects of prefetching affecting our read times. If we reach the beginning of our array, we wrap back around to the end of the array. Finally, we have a bounds check where we ensure that the time we have taken is between 0 and 1000000 ns. If not, then we set the time for this sample to 0 (the reason for this upper bound is described below in the analysis). Once we have completed 1000 loop iterations, we save the results for this array size and stride combination and then repeat until we have tested all array size and stride combinations.

We used very large arrays (maximum 1.5GB) because we needed to access data that was not cached in the L1 or L2 cache and that meant accessing data very far away from previous accesses. We know that only a certain amount of data is brought into a cache block on every cache miss. Therefore, at some point we will be accessing data in a large array that has not been brought into the cache along with previous data. This cache miss will result in a spike in the time to access the data but it will also bring in a whole set of new data in a cache block. Therefore, to continually have cache misses, we would need to continuously access data that was not brought in a previous cache block.

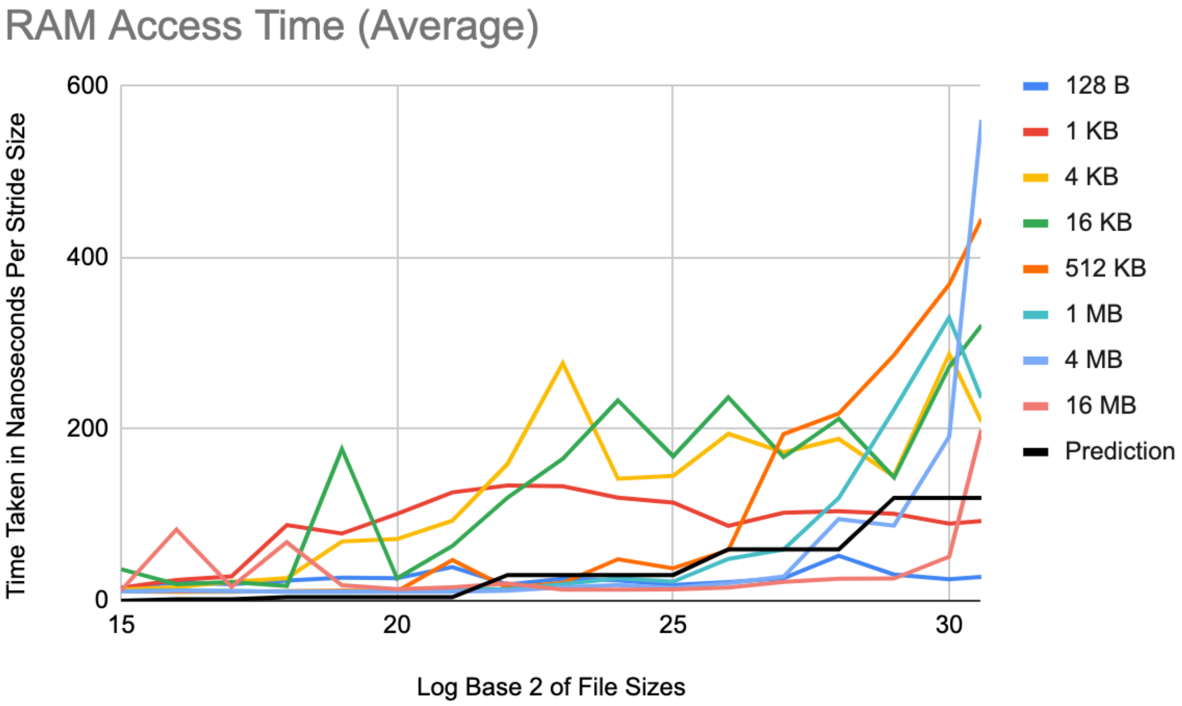
Prediction:

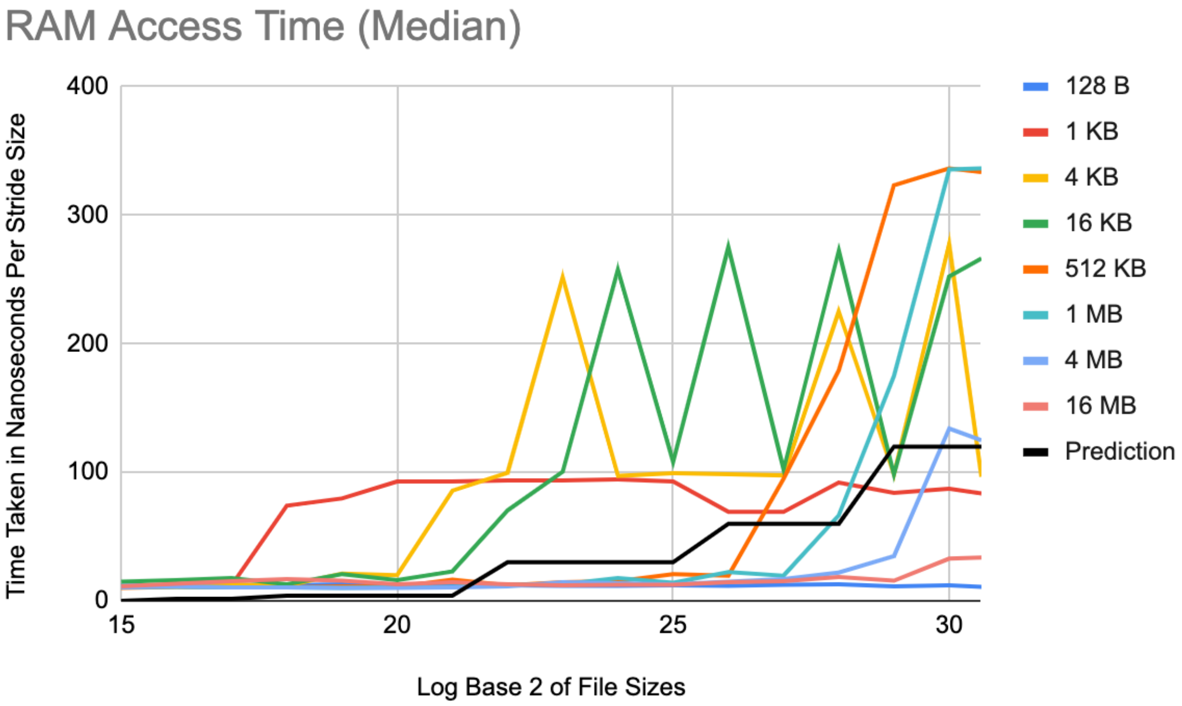
Base Hardware: Referring to this post and this article, we can see that for each level of memory, access time takes a different number of cycles. The cycle count and runtime is as follows:

- L1 Cache = 4 cycles (1.6 ns)
- L2 Cache = 10 Cycles (4 ns)
- L3 Cache = 40 ~ 75 Cycles (16 ns ~ 30 ns)
- Local DRAM = 150 Cycles (60 ns)

Software Overhead: We think that the only real software overhead we might experience in this experiment is if we access the memory but that causes a page fault. Then we would have to trap to the kernel and we would see a huge software overhead. However, this should not affect our results, because we are getting rid of extreme outliers in our results. We also know that there is a ~12ns delay for taking time measurements, therefore this overhead will be included in each measurement of RAM access. Since every operation has this delay, all times will be relatively larger by a small amount but not incorrect because all measurements are affected in the same way.

Results:


RAM Access Time (Average)
Time Taken in Nanoseconds Per Stride Size
600
400
200
0
15
20
25
30
Log Base 2 of File Sizes
128 B
1 KB
4 KB
16 KB
512 KB
1 MB
4 MB
16 MB
Prediction


RAM Access Time (Median)
Time Taken in Nanoseconds Per Stride Size
400
300
200
100
0
15
20
25
30
Log Base 2 of File Sizes
128 B
1 KB
4 KB
16 KB
512 KB
1 MB
4 MB
16 MB
Prediction

Analysis:

Our results are slightly different than those shown in the lmbench paper and from the values calculated using the links referenced. We can see that if we look at the average times against the stride lengths in the Average time graph, there is a constant growth in runtime throughout the different combinations of array sizes and stride lengths. Looking at the Median graph we can see an even clearer view of the sharp increases in runtimes. Although our results do not show the plateaus with each stride size as seen in the lmbench paper, we can definitely see a trend in which the time taken grows as we are entering different parts of the cache. We utilized the back-to-back load latency discussed in the lmbench paper by performing continuous cache miss reads from the file by traversing the arrays in backwards order to mitigate the prefetching effect and also use the stride sizes to index to different parts of the array. The 1KB stride length closely resembles the lines that are seen in the lmbench paper as it plateaus and stays constant throughout the rest of the experiment once it has reached a certain array size. One reason as to why our graphs take slightly longer compared to our calculations made in our predictions may be due to the implementation of reading the array backwards. This means we are not prefetching any of the data in the cache. It may not explain all the differences but may attribute to some of the results seen in our graphs.

We determined that samples that had times greater than 1000000 ns were invalid for measuring the overhead in performing a RAM access. The upper bound allows us to remove the outliers that occur due to page faults since we are only trying to measure RAM accesses. We were initially seeing these outliers in our results and skewing our data, therefore at a later stage in the experiment we added the upper bound to combat this noise.

Note that we didn’t use a standard deviation metric in this example as it was not useful in understanding our results.

b. RAM Bandwidth:

Methodology:

We began by mallocing a 512MB array of *chars* in memory in order to create a large array that we could read and write from. We then replaced each byte in the array with a random *char*. We note that this will cache the array in memory. This is not a problem since we are testing the bandwidth of RAM and do not want to perform page faults to the disk. This test then has 2 parts to it, reading and writing to RAM, where the read test is performed first, then the write test.

Test 1: Read

We first sample the current time as the start time. We then create a new *x* byte array of *chars*, where *x* is the number of loop unrolled instructions we are using, that we will use to store the data we read in from memory. Next, we begin our loop unrolling. For each loop iteration, we read *x* bytes of memory from the start of each *x* byte chunk in memory one by one and save the byte read into our *x* byte array of *chars*. Once we have read the complete array by completing all the for loop iterations, we take the current time as the end time. Note that *x* will take on the values 16, 32, and 64 and the results and analysis for this decision is below.

Test 2: Write

We first sample the current time as the start time. Since we already have a 512MB array of *chars* in memory, we used that as the memory we will be writing to. Next, we begin our loop unrolling. Following the same logic from the read tests, we write to memory in *x* byte chunks using *x* loop unrolled instructions. Each loop unrolled instruction writes the *char* '1' to a byte in memory. For each loop iteration, we write *x* bytes of memory from the start of each *x* byte chunk in memory and therefore write each byte sequentially from the array in *x* byte chunks. Once we have written the complete array by completing all the for loop iterations, we take the current time as the end time. Note that *x* will take on the values 16, 32, and 64 and the results and analysis for this decision is below.

Then, using the total time taken for each test, we converted the time taken in nanoseconds to seconds, converted to MB/sec by dividing 512MB by the total amount of seconds taken, and then finally dividing that GB to obtain GB per second. This provides us the bandwidth of the reading and writing operation.

We then run this experiment 100 times using a bash script to record all of our results and calculate the average and median times for each number of loop unrolled instructions (16, 32, 64) for the read test and the write test.

Prediction:

Base Hardware: Using the following articles: article1, article2 we calculated the memory bandwidth. We used the Memory Clock Frequency and the Memory Bus Width which we already have from the machine description to do the calculation of the memory bandwidth:

$$2133\,Mhz \times (64\,bit\,/\,8\,bit) \times 2 = 34.128\,Gb/s$$

We would expect the Memory Read bandwidth to be larger than the Memory Write bandwidth as generally speaking there is roughly a 3 times performance difference between memory read and memory write.

Software Overhead: We expect that in this experiment the only potential overhead from software will be the overhead from executing a minimal loop. This is because we are using loop unrolling in a loop, and the loop itself could add a small overhead (roughly 2μ*s* from results in test 1a.ii). We also know that there is a ~12ns delay for taking time measurements, therefore this overhead will be included in each sample after the for loop completes. Since every sample has this delay, all times will be relatively larger by a small amount but not incorrect because all measurements are affected in the same way.

Results:

Read Bandwidth Results

	Average GB/sec	Median GB/sec	Std Dev.
16 loop unrolled inst.	15.892250559337	16.411738182249998	0.9255905510168052
32 loop unrolled inst.	40.053863536277	40.8663837599	6.186587663145737
64 loop unrolled inst.	103.47845110131202	108.9466514456	17.789285973034282

Write Bandwidth Results

	Average GB/sec	Median GB/sec	Std Dev.
16 loop unrolled inst.	19.183352983218	19.50367535525	0.8569416828962045
32 loop unrolled inst.	52.30264761851199	54.46222853535	6.784481928318065
64 loop unrolled inst.	119.89727329285803	124.8807772152	18.70301826034899

Analysis:

Our results are very different from what we predicted to see. As shown in the table, as we varied the number of loop unrolled instructions our bandwidth for both reading and writing increased significantly. Another surprising result is that all measurements for the read bandwidth are lower than the write bandwidth which should not be true according to our predictions. Interestingly though, the standard deviation across all tests are relatively small which means those numbers were very repeatable during our sampling process. Let us discuss why we think we are seeing these results.

Today's memory operations and hardware in general are very complex and have a number of hidden optimizations. Since we used a bash loop to gather our samples, this means that a new process is spawned for every sample. What this also means is that a new 512 MB array of *chars* is malloced onto the heap for every process that runs. However, since we fill the array with chars before performing the tests we know that main memory will contain the data for the array. When reading from the array we are able to read so much data so quickly from the cache that we are seeing minimal time needed to read a lot of data. The same is true for writing. This explains why our speeds are so high, but does not explain why we see so much variability in the difference between loop unrolled instructions.

We believe that our calculations for bandwidth as a throughput measure are correct, therefore we are not worried about our conversions to calculate the final output metrics. Therefore, we believe that the variability that arises from the difference in loop unrolled instructions comes from the fact that the instructions in the loop are independent of each other and therefore are executed in parallel. This parallelism is performed by the hardware and will allow for major performance improvements. We also remove more of the overhead from the loop and missed branch predictions as we increase the number of loop unrolled instructions (note that in an earlier experiment we saw that the loop overhead for 1 minimal loop iteration is $\frac{1983.2096\,ns}{1000\,iterations} = 0.1983ns$).

Another hardware optimization that we believe could be happening is cache line prefetching. Since we are continually reading from the same block of memory sequentially, the hardware could be bringing data from main memory all the way into the L1 cache before we access it allowing there to be major improvements in read and write time. Modern machines today focus on performance and use aggressive caching to do this, therefore we believe that this is the reason for the variability in our results.

We attempted to perform the test with other methods for read and writing from memory using library functions like *memset*, *memcpy*, and *bcopy* but we decided to use loop unrolling to avoid the overhead of those library functions.

c. Page Fault Service Time:

Methodology:

In this experiment we are timing how long it takes for the system to service a page fault. In order to cause a page fault, we first flush the main memory to remove any of the physical pages that contain the data we want to access because having that data in physical memory would alter our measurements since no page fault would occur. We first malloc sixteen 1GB arrays of *chars* and for each array we fill it up with random characters. By accessing every byte in every 1GB array, we will completely flush out any data that was in the 16GB main memory before.

Logically, this occurs because we are accessing data malloced into the processes virtual address space that is not currently in the machine's physical memory. Therefore, accessing the malloced data will cause page faults to bring that data into the machine's physical memory for writing.

At this point main memory's physical pages should only contain data from the 1GB arrays. We then memory map a file into the current processes' address space by running the *mmap* procedure. This procedure will give us a pointer to the location in memory that stores the data for that file. Note that *mmap* only maps the file to the process' virtual address space, and therefore the file data is not in any physical page yet. Next, we access an entire page, which is 4096KB, that was mapped into the process' virtual address space. The first access will cause a page fault to occur because we have completely flushed our main memory, meaning physical memory should only contain the data from the 1GB arrays. Then all reads for the rest of the pages should be reading from memory. The code that we measure is the code used to access one entire page of data from the memory mapped file because that memory mapped file is in virtual memory but not physical memory, and therefore will cause a page fault on the first access only. We then use a bash loop to take 10 samples of page fault service time, and calculate the average across those as our average page fault service time. We would have liked to gather more samples however the operation of mallocing and writing a total of 16GB per sample was a very time-intensive operation.

Prediction:

Base Hardware: After doing some research to get an idea of how long a typical page fault service time takes, we came across the following Wikipedia page, which considers a page fault to be a function of rotational latency + seek time. They estimate that although these latencies differ from machine to machine, on average, a typical machine takes around *5ms* for rotational latency and *3ms* for seek time. This results in a page fault service of *8 ms*.

We feel that this number is very large (especially since our PC has an SSD with no disk heads). Since the machine we are using has SSD memory, we have no rotational latency. Furthermore, we expect our seek time to be much less (maybe 100x times faster) than spinning disks. We predict this number will be more on the order of 10s of microseconds.

Software Overhead: In order to test the page fault service time, we are accessing a whole page of data from our buffer in a loop. The software overhead that could be added here is the loop overhead measured in experiment 1a. ii (roughly $2\mu s$). However, the loop in that test ran for 1000 times, whereas this experiment will run for 4194304 times (about 4000x longer) so perhaps some 5-8 ms of our measured results are just the loop overhead. We also know that there is a ~12ns delay for taking time measurements, therefore this overhead will be included in each measurement of page fault service time. Since every measurement has this delay, all times will be relatively larger by a small amount but not incorrect because all measurements are affected in the same way.

Results:

Average Page Fault Service Time	36767093.92 ns = 36.76709392 ms
Median Page Fault Service Time	30849180.0 ns = 30.8491800 ms
Average Page Fault Read Time per Byte	8.765958290100098 ns
Std. Dev Page Fault Service Time	23656235.130997702 ns

Analysis:

Our results show that the average page fault service time is 36 ms (36.76 ms), however there is a lot of variability in our data. This is in line with our prediction and we believe this is a valid measurement because page faults mean going to disk and that is an operation that takes time in the order of milliseconds.

Using the command *getconf PAGESIZE*, we see that MacOS has a page size of 4096KB. Therefore dividing the average page fault service time by the size of a page, we get:

$36767093.92\, ns \,/\, (4096 \times 1024)\, bytes \;=\; 8.765958290100098\, ns/byte$

which is the average time to access a byte including the time needed to service a page fault. When we compare this result to the latency of accessing a byte from main memory, which takes 15 ns/byte, we see that it is lower which is not what we expected. We believe that we are seeing a low average due to caching effects and prefetching by the underlying system through memory operations. Also it is important to note that only the first byte access will cause the page fault and the rest should be read from memory (actually the L1 cache). Therefore, the effect of the 1 byte that had a large latency is hidden when we calculate the per byte average.

SSDs are very optimized for certain tasks, and SSDs are 1000x faster than hard drive disks. We know that the hardware we are performing is very strong, but this test just goes to show how slow disk accesses are.

As mentioned earlier we were only able to run 10 samples of this experiment because it took a long time due to the need to flush RAM before each experiment.

Network

a. Round Trip Time:

Methodology:

For this experiment we have two C files, one of which contains the client code, and one of which contains the server code.

The client code is what contains most of the functionality for this test. We always run the client code on our local machine. The client will create a socket, choose the IP address of the server to connect to based on the test we are doing (local or remote), and then connect to the server on port 5000. Now that we have connected to the correct server we begin our experiment. In a for loop, we execute a "ping" operation many times. We coded a ping operation as sending a 64 byte zero-initialized buffer to the server and immediately waiting to receive it back. In our for loop, we calculate the start time, execute a ping operation, then calculate the end time providing us with the total round trip time for this sample. We then perform this process 100 times to get a stable average round trip time.

The server code is written to be very simple. The server will open a new socket, bind to the socket on port 5000, and then listen on the socket. It then will infinitely loop to wait for a client connection request and accept it once one comes in. Then we have hardcoded the number of "pings" we will send to the server, so we have a for loop that will execute *recv* and *send* in that order for the number of ping times. This allows us to receive a message (which we have specified as 64 bytes) and then immediately send that message back to the client for multiple samples. No timing measurements are taken on the server. This server code can be ported directly from the localhost onto our remote machine and run the same way.

To run the test for localhost to localhost we specify the server address as *127.0.0.1* and for localhost to remote host, we specify the server address as the ip address of our Amazon EC2 Server.

Prediction:

Base Hardware: The network interface cards of both the local machine and the Amazon EC2 server are used to receive and transmit packets of file data across the network. Both of these network cards have a max bandwidth at which they can transmit data to and from the network. Therefore, the performance of these cards could limit the performance of the remote file data transfer.
We estimate that the overhead of the NIC receiving and processing the packet depends on the Network card bandwidth which in our case is 1300 Mbps, thus,

the overhead will be the amount of data we are sending (64 byte) divided by the bandwidth: $64\ byte\ \times \frac{1\ sec}{1300\ MB} \times \frac{1\ MB}{1048576\ Bytes} \times\ 10^{9} \frac{ns}{s}\ =\ 46.95012\ ns$

Software Overhead: In this test, there is also a network penalty involved since we must now wrap packets with the correct TCP/IP protocol, and transmit them through the network interface card. The same operations must then be performed in reverse to receive the data from the network. In the localhost to localhost case, according to our research, when we trap to the OS to send the packet, the OS will identify it as sending the data to itself and immediately return it back to the application. Therefore, we predict the network penalty to be low in this case as we do not actually have to send data over the network. We also know that there is a ~12ns delay for taking time measurements, therefore this overhead will be included in each measurement of round trip time on the network.

We ran the ping command on our local machine and saw that pinging the localhost from our local machine had the following results. We expect our results to be very similar to 0.103 ms, which equates to 103000 ns, because we are implementing the same functionality of sending and receiving one packet from client to server.

```
--- 127.0.0.1 ping statistics ---100 packets transmitted, 100 packets received,
0.0% packet lossround-trip min/avg/max/stddev = 0.043/0.103/0.976/0.122 ms
```

We ran the ping command on our local machine and saw that pinging the remote server from our local machine had the following results. We expect our results to be very similar to 106.464 ms, which equates to 106464000 ns, because we are implementing the same functionality of sending and receiving one packet from client to server and will receive the same network penalty as the *ping* command.

```
--- 18.208.180.236 ping statistics ---
100 packets transmitted, 99 packets received, 1.0% packet loss
round-trip min/avg/max/stddev = 82.689/106.464/495.338/61.463 ms
```

Results:

Localhost

Average Time Taken	221649.920000 ns
Median Time Taken	193440.800000 ns
Std Dev.	233854.52007818536

Predicted Time Taken	103000 ns

Remote

Average Time Taken	86696686.912000 ns
Median Time Taken	85589230.400000 ns
Std Dev.	29953126.05603457
Predicted Time Taken	106464000 ns

Analysis:
Our results for an application level ping from localhost to localhost are approximately 100,000 ns slower than the results for a ping from localhost to localhost using the *ping* command. We attribute this difference to the fact that the *ping* command is using ICMP requests which are handled at the kernel level and not the application level. Therefore, those requests are immediately sent back to the client without ever needing to cross the protection domain from kernel to application. In our application level ping, we perform the following protection domain crossings:
client app → localhost kernel → localhost kernel → server app → localhost kernel → localhost kernel → client app.

This shows that we cross the protection domain boundary 4 times to send a message from the client and receive the return message from the server. Compared to the *ping* command, which services ICMP requests at kernel level, there will only be two protection domain crossings and those are between the client and the localhost kernel. The server application is never involved.

Our results for an application level ping from localhost to a remote server are approximately 19,767,313 ns faster than the results for a ping from localhost to a remote server using the *ping* command. We assume that the *ping* command has other overheads such as DNS that may cause an increase in the latency for ping requests. However, it is valid to note that the time for pinging a remote server was many orders of magnitude larger than for pinging the local host for both our application level ping and the *ping* command.

In both experiments, we see a large standard deviation but this is expected. Using the network always introduces variability as the network is very chaotic. Packets may need to be re-sent, the network may have a lot of traffic, or the machine may be under a lot of stress. All of these factors could introduce variability in our measurements.

b. Peak Bandwidth:

Methodology:

For this experiment we have two C files, one of which contains the client code, and one of which contains the server code.

The client code is what contains most of the functionality for this test. We always run the client code on our local machine. The client will first create a socket, and then choose the IP address of the server to connect to based on the test we are doing (local or remote), and lastly connect to the server on port 5000. Now that we have connected to the correct server we begin our experiment. We first initialize a data array to 256 bytes to use this as the data to send over the socket to the server. We chose to send data in 256 byte chunks because this would allow us to simulate a streaming interface for sending data over the network link to the server. Next we specify the total amount of data to send to the server as *x* GB (our sent data size varied to show varying results, this is explained in the analysis section) as this should be enough to saturate the network link and see our peak bandwidth. Next, for 100 samples we do the following: get the current time as the start time, in a while loop send 256 byte chunks of data over the socket until we have sent a total of *x* GB, and then get the current time as the end time. Once we have gathered all the time taken for each sample, we close the socket and calculate the bandwidth metrics in terms of bytes/sec, MB/sec, GB/sec, and Gbits/sec.

The server code is written to be very simple. The server will open a new socket, bind to the socket on port 5000, and then listen on the socket. It then will infinitely loop to wait for a client connection request and accept it once one comes in. Once a connection request arrives, a while loop begins that continuously receives data that until the amount of data the client stops sending data (in this case *x* GB). No timing measurements are taken on the server. This server code can be ported directly from the localhost onto our remote machine and run the same way.

To run the test for localhost to localhost we specify the server address as *127.0.0.1* and for localhost to remote host test, we specify the server address as the ip address of our Amazon EC2 Server. Note that we also confirmed the bandwidth for the server is higher than the local machine and therefore we will truly test the throughput for sending data over the network.

Prediction:

Base Hardware*:* The only hardware that is directly involved in our timings for this experiment is the Network Interface Card (NIC). The NIC will be responsible for sending packets. This exact overhead depends on the amount of data we are sending, but once we know that we can use the latter half of the equation shown in the round trip time calculations to find this overhead.

Software Overhead: From CSE221 this quarter, we learned that the cloud has high latency and low bandwidth. We are using an Amazon EC2 instance which is a cloud machine. Therefore, we expect the network penalty to be high as we try to send more data to the server. We also know that our server is located in North Virginia, therefore the distance the packets must travel is very far. We expect our network penalty to increase with the amount of data we are sending. From our Network Experiment A, we saw that the round trip time for 1 packet was 86696686.912000 ns which is equivalent to 86 ms. Therefore we will multiple 86 ms with the number of packets we send which are of size 256 bytes. Mathematically this is $(data\ size/256\ bytes)\ \times\ 86\ ms$ approximately which is a very large overhead. We also know that there is a ~12ns delay for taking time measurements, therefore this overhead will be included in each measurement of Network peak bandwidth.

To have a benchmark to compare our empirical results with, we used a tool to measure the maximum achievable bandwidth on IP networks called *iperf*. We ran the *iperf* command to localhost in order to measure the bandwidth in a loopback interface and we saw that the average was 29.9Gbits per second. We also ran the *iperf* command from localhost to remote host in order to measure the bandwidth to the remote interface and we saw that the average was 31.2 Mbits per second.

Since we are conducting the same experiment of timing the operation of sending the entire file and calculating the bandwidth we would expect our results to be similar.

	Interval in seconds	Transfer size in GB	Bandwidth in seconds
Localhost to 127.0.0.1	0.0 - 10.0 sec	34.8 GBytes	29.9 Gbits/sec
Localhost to Remote host	0.0 - 10.2 sec	37.9 MBytes	31.2 Mbits/sec

Results:

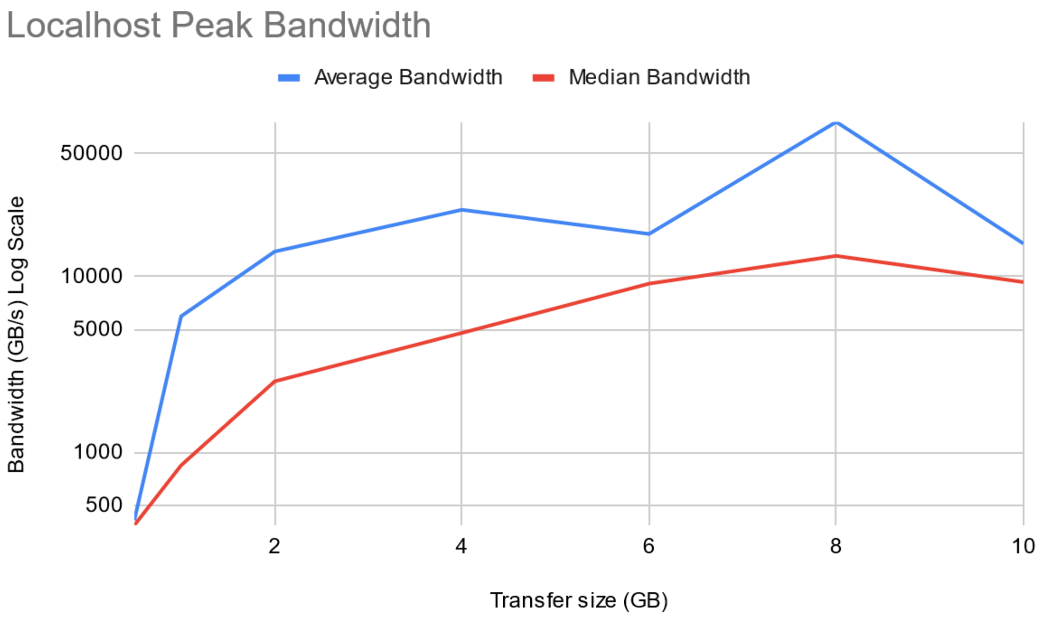
Localhost to 127.0.0.1

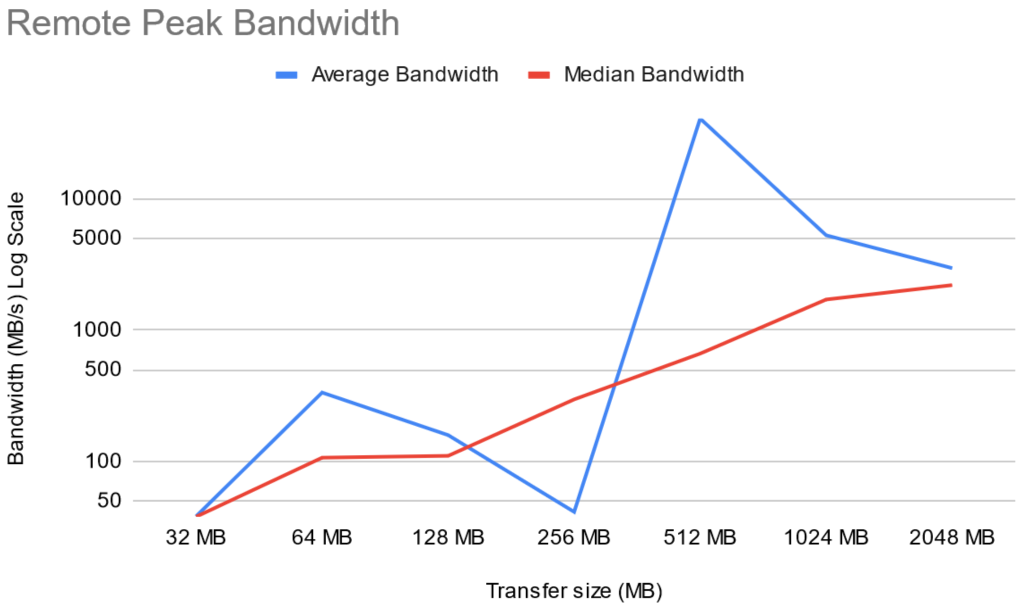
Transfer size in GB	Average Bandwidth	Median Bandwidth	Std. Dev
0.5 (100 samples)	411.60003 MB/sec	385.55867 MB/sec	166.35183 MB/sec
1 (100 samples)	5954.34948 MB/sec	844.90041 MB/sec	18099.44999 MB/sec
2 (20 samples)	13896.21993 MB/sec	2538.49017 MB/sec	35876.40512 MB/sec
4 (10 samples)	24019.61066 MB/sec	4785.50701 MB/sec	53242.04703 MB/sec
6 (10 samples)	17498.04589 MB/sec	9125.13749 MB/sec	21071.99067 MB/sec
8 (10 samples)	108529.42285 MB/sec	7125.93147 MB/sec	142991.66995 MB/sec
10 (10 samples)	15418.52693 MB/sec	9313.70769 MB/sec	15129.30231 MB/sec

Localhost to Remote host

Transfer size in MB	Average Bandwidth	Median Bandwidth	Std. Dev
32MB (10 samples)	38.27136 MB/sec	37.99042 MB/sec	11.04037 MB/sec
64MB (10 samples)	334.88487 MB/sec	106.98267 MB/sec	704.59183 MB/sec
128MB (10 samples)	158.87030 MB/sec	110.47535 MB/sec	96.72235 MB/sec
256MB (10 samples)	41.36245 MB/sec	296.94640 MB/sec	159.65345 MB/sec

512MB (10 samples)	40775.47702 MB/sec	662.82791 MB/sec	118455.46537 MB/sec
1024MB (10 samples)	5268.90420 MB/sec	1710.22834 MB/sec	10785.60389 MB/sec
2048MB (10 samples)	2962.65423 MB/sec	2201.57866 MB/sec	1272.88572 MB/sec





Analysis:

First, we will begin our analysis of the Localhost to 127.0.0.1 results. We see here that the bandwidth measurements are increasing in MB/sec transferred as we transfer larger amounts of data. However when we reach 6 GB and on, we see that all bandwidth measurements are ~10000 MB/sec. This means that we have found the peak bandwidth for the Localhost to 127.0.0.1 data transfer. For larger amounts of data, the machine becomes bottlenecked by the amount of data it needs to send and how much it can actually send over the network.

Next, we will analyze the results for Localhost to Remote host. We were unable to perform this test using the data transfer sizes that we used for localhost to localhost. The time required to send data to and from the remote server was much longer, though our results don't reflect this because those are not the total times for the program. Using the results we gathered, we see a lot of variability in our measurements. This is due to the fact that the connection to the remote server includes the network and the network itself is very volatile. We are also using a free tier server (*t2.micro*) from Amazon AWS, therefore our machine is not meant to provide superior performance. Third, the remote machine is located very far away from Northern California where the tests are being performed. Looking at the results, we can see that the median bandwidth increased as we increased the amount of data transferred, however it begins to plateau at ~2500 MB/sec. We believe that this is the peak bandwidth over the network for the machine.

Comparing the results of the two connections, we see that the remote test has much lower bandwidth metrics than the remote host. This is exactly what we expected because the network penalty of sending data will require the remote test to use more time to send the same amount of data as the localhost test. Therefore the remote bandwidth will decrease. It is also interesting to see how much more variability occurs with the remote test compared to the localhost experiment.

In both experiments, we see a large standard deviation but this is expected. Using the network always introduces variability as the network is very chaotic. Packets may need to be re-sent, the network may have a lot of traffic, or the machine may be under a lot of stress. All of these factors could introduce variability in our measurements.

c. Connection Overhead:

Methodology:
For this experiment we have two C files, one of which contains the client code, and one of which contains the server code.

The client code is what contains most of the functionality for this test. We always run the client code on our local machine. We first initialize the server ip address as the localhost or remote and then specify port 5000 as the port we want to connect to. In a *for* loop, the client will first create a socket to connect to the server on port 5000. It will then perform the set up operation by calculating the start time, executing the *connect* procedure to connect to the server, and then calculating the end time to provide us with the total time taken to set up the connection. Next we sleep for 1 second to allow the connection to stabilize. We then perform the tear down operation by calculating the start time, executing the *close* procedure on the socket used to connect to the server, and then calculating the end time to provide us with the total time taken to tear down the connection. We repeat this process for 100 samples to get a stable statistical measure for both operations.

The server code is written to be very simple. The server will open a new socket, bind to the socket on port 5000, and then listen on the sock. It then will infinitely loop to do the following. It will wait for a client connection request and accept it once one comes in. No timing measurements are taken on the server. This server code can be ported directly from the localhost onto our remote machine and run the same way.

To run the test for localhost to localhost we specify the server address as *127.0.0.1* and for localhost to remote host, we specify the server address as the ip address of our Amazon EC2 Server.

Prediction:
Base Hardware: The network interface cards of both the local machine and the Amazon EC2 server are used to receive and transmit packets of file data across the network. In this case, this hardware should provide only minimal penalty as we are sending only small data packets over the network. We cannot know the exact overhead because we do not know how much data the machine will send to establish a connection, however, For the exact measurement formula of the NIC overhead, please refer to the calculations done for round trip time.

Software Overhead: There is a small network penalty for this test to wrap data in the TCP/IP protocol, encrypt it, and transmit it through the NIC. The same operations must then be performed in reverse to receive the data from the network. However, since we are sending minimal packets and only once for *connect* and once for *close*, the overhead should be minimal. In the localhost to localhost case, according to our research, when we trap to the OS to send the packet, the OS will identify it as sending the data to itself and immediately return it back to the application. Therefore, we predict the network penalty to be low in this case as we do not actually have to send data over the network. We also predict the network penalty to be low for the localhost to remote host case as we are sending minimal data. We also know that there is a ~12ns delay for taking time measurements, therefore this overhead will be included in each measurement of connection setup and teardown time.

We predict that the time taken to set up the connection will take 10,000s of ns longer than to tear down the connection. The reason for this is because setting up the connection will need to have acknowledgement from the server if we actually connected and this could take longer. However, the tear down operation is one sided in that it closes the connection on the client side and the server will react on it’s own time to handle the closed connection. Therefore this is quicker. However, these operations require kernel support so we believe crossing the protection boundary to set up and tear down the socket will take 10,000 ns and therefore these are long operations.

Results:
Localhost

Average Set Up Time	573685.328000 ns
Median Set Up Time	343928.800000 ns
Std Dev. Set Up Time	119640.589040 ns
Average Tear-down Time	75729.408000 ns
Median Tear-down Time	51800.800000 ns
Std Dev. Tear-down Time	27034.683077 ns

Remote

Average Set Up Time	95011011.536000 ns
Median Set Up Time	93587943.600000 ns
Std Dev. Set Up Time	395352.401160 ns
Average Tear-down Time	136107.120000 ns
Median Tear-down Time	105210.400000 ns
Std Dev. Tear-down Time	33702.944835 ns

Analysis:

As we predicted, the setup time usually takes longer than tear down time. This is because in order to set up we need to send a packet to the server and wait for an acknowledgement. We can see that our average set up time for remote servers is actually roughly the same as our ping time, which further supports this claim.

We predicted that this time will be 10,000s of nanoseconds higher but in our results we see that it is only 1,000s of nanoseconds higher. This can be because we overestimated how fast tear down happens. According to GNU Documentation we see that when we close the socket, we still perform many operations and cleanup code. This can take some time.

Another trend that we see is that local connection setup takes much longer than remote connection set up, which makes sense, because we now have to wait for the transmission of packets over remote channels. But we also see that tear down times are roughly the same for both remote and local connections. This also makes sense because we know that tear downs are mostly cleanup that happens on the client side, so we would expect that the local and remote connections both have similar tear down time.

In both experiments, we see a large standard deviation but this is expected. Using the network always introduces variability as the network is very chaotic. Packets may need to be re-sent, the network may have a lot of traffic, or the machine may be under a lot of stress. All of these factors could introduce variability in our measurements.

d. Networks Experiments Comparison:

Evaluate for the TCP protocol.
In our Network experiments, we utilized the TCP Protocol to send data across the network. TCP is designed to break the data sent from a client into packets that can be pushed over the network to the receiving server. We implemented our client-server communication using the TCP/IP API in C (*socket(), connect(), listen(), accept(), recv(), send(), close())*.

Comparing the remote and loopback results, what can you deduce about baseline network performance and the overhead of OS software?
In all of our network experiments, we saw that the loopback interface performed much more favorably than our remote interface. These results demonstrate that even though the OS is not as efficient as application level, the network provides even less efficiency. We saw this during CSE 221, since networked operating systems such as the V Kernel exhibited lots of inefficiencies due to the network penalties it faced. In certain cases, such as the data center, the network penalty may be acceptable due to the high bandwidth and the high latency of the network. However, we used the cloud for our experiments and proved that the network penalty was unacceptable when compared to the overhead of operating system software.

For both round trip time and bandwidth, how close to ideal hardware performance do you achieve?
For our round trip time experiment, we saw that the overhead for localhost was 0.22164992 ms and the overhead for the remote host was 86 ms. Ideally, we would like the remote host to perform as well as the localhost or better, but we see here that there is an extreme performance difference (~85 ms) between the network and ideal hardware performance. This is a large enough difference where the performance degradation is visible to the user.

For our bandwidth experiment, we saw that the bandwidth for localhost was ~10000 MB/sec and the bandwidth for remote host was ~2500 MB/sec. This explicitly shows that the network limits the performance of sending and receiving data to very small amounts. Ideal hardware performance would be above 10000 MB/sec, and we are achieving only 25% of that performance when using the network.

What are reasons why the TCP performance does not match ideal hardware performance (e.g., what are the pertinent overheads)?
TCP performance does not match ideal hardware performance for many reasons. First reason is that it must first chunk the data into packets and then wrap them in the TCP/IP protocol. We will need to do this for each packet and there may be many packets being sent so this overhead which is compounded.

Next is in order to send packets, we must interact with the kernel and the NIC to access the network and crossing the protection domain to do this is expensive. TCP also uses retransmission to ensure that packets reach their destination and this is also expensive if we need to retransmit frequently. On the receiving end of the TCP packets, the machine must unwrap the packets to access the data as well as piece together the data before actually acting on it. These are all overheads in TCP that explain why TCP performance does not match ideal hardware performance.

File System

a. Size of File Cache:

Methodology:

We begin this experiment by creating a 16GB file in our directory that we can use in order to read from and complete our experiments. The file is a generic text file filled with random characters and was created before running our experiments. We reuse the same 16GB text file for each file size in the experiment which means that as we vary the file size, we will still be reading data from the same file. This should eliminate any discrepancies that may arise from running the experiment as well as display the increase in time taken as the amount of data from the file increases. We initialize a 64KB buffer that we use to read the file in chunks so that we avoid filling up the heap. This is similar to a streaming interface. If we didn't use this method of reading data, we may cause situations in which we would need to page to the disk and we want to avoid this. We initialize the amount of data to read, or file size, to be 0.5GB at the beginning of the experiment, and increment by 0.5GB each iteration. We continue this process for 32 file sizes in order to get to a final file size of 16GB, meaning we are reading the entire file. Each of these different size reads will also be done 16 times each in order to get an average time that would better illustrate the size of the file cache. This is to amortize any startup effects when reading the files.

It is important to note that since we are testing the size of the file buffer cache, every time we read for a new file size, we start at byte 0 in the file. Therefore all of the data that was in the file buffer cache on the previous read will still be there. This will allow us to see the increase in time taken to read more data from the file buffer cache as well as when the file buffer cache no longer contains data that is being read in.

We performed this experiment after rebooting the machine to avoid any issues with the caches' state when we started the experiment.

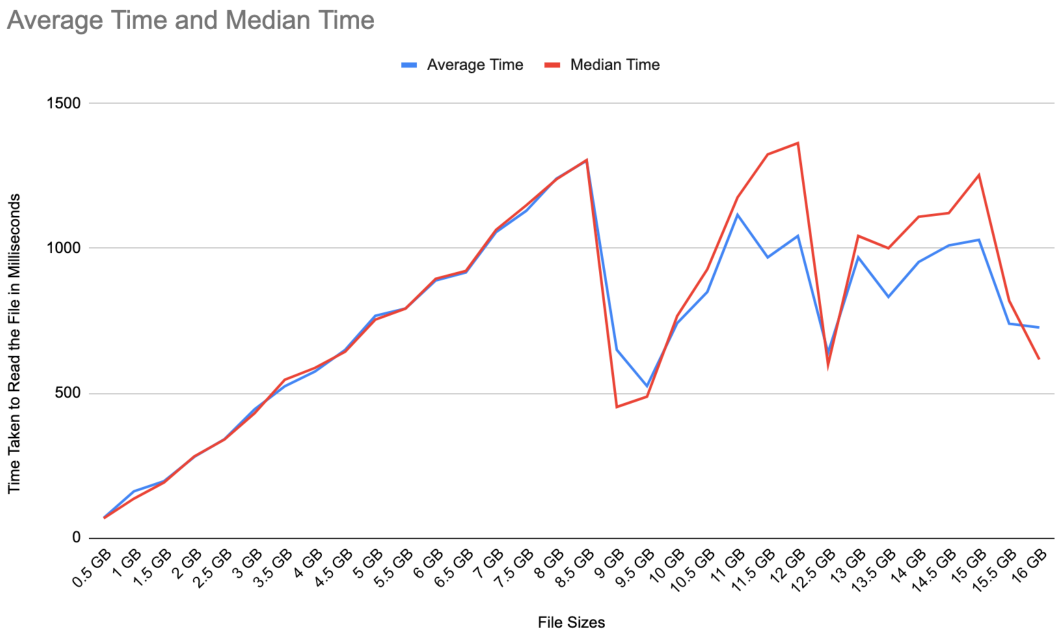
Prediction:

Base Hardware: We are interacting with both main memory and disk in this experiment. According to our research, accessing the disk will require 1-3 ms and from our Memory Operation A experiment we know that accessing memory has a 12 ns delay. We also must take into account the disk transfer rate when we page fault since this will affect how fast we can read data from the disk. The page fault service time is described further in the Software Overhead section.

Software Overhead: When we page fault, there will be extra overhead that we measure because we are now performing a disk read. When performing a disk read we must execute the *read* syscall, transfer control to the kernel to handle the disk I/O, and then copying that data into main memory for the process to access. This is all done in software and we know that syscalls can take ~4000 ns which is relatively large. From our memory operation C the time it takes for software to service a page fault is 36 ms, therefore the overhead for reading will be very large when we page fault. We also know that there is a ~12ns delay for taking time measurements, therefore this overhead will be included in each measurement of FBC.

After some research, we found out that the File Buffer Cache (FBC) grows dynamically and that modern OSes aggressively cache file data. Based on this knowledge, we predict that the time it takes to read a file grows linearly as a function of file size. But there are spikes at certain sizes (The exact number of these sizes are very hard to predict since they are dynamically allocated and depend on the specific hardware we are using). These spikes should occur as we read larger and larger files, because the OS will just allocate a larger chunk of main memory to the FBC which causes file data to be cached and that is why the read times will suddenly become very fast even for large files. After the decrease in read time, it will once again grow linearly as a function of the file size until we hit the next threshold where the OS will increase the FBC size again and we will see another spike and so on.

Results:



Analysis:
As we predicted, the graph above shows a linear growth. This means that as we increase the size of the file, it takes longer to read the contents of that file. However, we see that we also have 3 peaks, followed by 3 drops. As we predicted, these peaks and drops are due to the OS allocating more pages of physical memory for the FBC which allows the process to just aggressively cache the entire file. Thus, after this increase in the size of FBC we see a huge drop in the time taken because the file content is now cached and takes less time to read. The FBC uses prefetching for file data to exploit spatial locality because it wants to avoid FBC misses by storing data that will be accessed in the near future into physical memory. Then we see the same pattern again and again. There is a linear growth in the time taken as the size of the file increases, and then at a certain point, the OS increases the size of FBC and we see a drop since the file data is cached.

We can empirically see that these peaks happen at *8.5GB, 12GB,* and *15GB,* for the machine we are using. Again these sizes are dynamically allocated by the OS and depend on how much main memory is available, how many other processes are running and many other factors.

Note that we didn't use a standard deviation metric in this test as it was not useful in understanding our results.

b. File Read Time:

Methodology:
To compare the time it takes to read a file sequentially versus randomly as a function of file size, we performed the following experiment. We decided to use a 16GB file to determine how our read times fluctuated as file size increased. To avoid caching effects and explicitly read from the disk when performing I/Os, we opened the file using the *F_NOCACHE* and the *O_RDONLY* flags (we discuss the issues with this in the analysis section). We then malloc a buffer that is the size of a block (4096 bytes) in the file system on our machine (used the command *diskutil info / | grep "Block Size"* to confirm this).

Now we begin our reading tests. For each file size, we increment in sizes of 0.5GB (0.5GB, 1GB, 1.5GB, ..., 16GB). We perform both sequential reads and random reads. Note that because we are performing these I/Os directly from the disk, we do not need to worry about cache effects (not necessarily true, we expand on this in the analysis section) for reading the same data in and more every new file size or for reading the same file twice (sequentially then random). Once we receive the total time taken to perform the complete I/O for each file size, we store the results in a results file.

The sequential read operation is performed as follows: First, seek to the first byte of the file. Next, read the file sequentially in 4096 byte chunks (time how long it takes to read each chunk from the file) until the complete file is read. Finally return the average per-block read time.

The random read operation is performed as follows: In a loop, we first select a random offset in the file that is not within 4096 bytes of the start or end of the file. Next, we seek to that byte offset of the file, read 4096 bytes from that offset (time how long it takes to read the chunk from the file), repeat this process until we have read the specified file size number of bytes. Finally, return the average per-block read time.

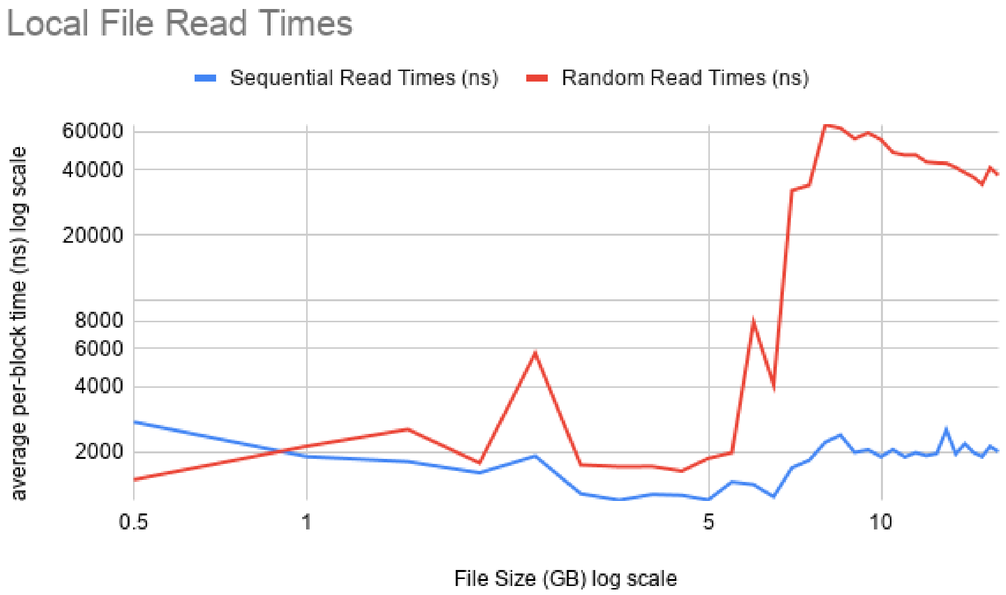
Prediction:
Base Hardware: We are interacting with the disk in this experiment. According to our research, accessing the disk will require 1-3 ms. We also must take into account the disk transfer rate when we read from the disk as this will dictate how fast we can perform our reads. It is important to note that we could not find an

effective way to turn off the FBC in MacOS, therefore the disk penalties may not appear in our results.

Software Overhead: When performing a disk read we must execute the *read* syscall, transfer control to the kernel to handle the disk I/O, and then copying that data into main memory for the process to access. This is all done in software and we know that syscalls can take ~4000 ns which is relatively large. We also will see an increase in the number of instructions executed relative to how many times we iterate in our loop to read by block size. The loop will incur penalties in our time as well as the timing mechanisms we use. Another software overhead that could be added here is the loop overhead measured in experiment 1a. ii (roughly $2\mu s$). However, the loop in that test ran for 1000 times, whereas this experiment will run for the number of blocks in the file. That is each sample will run for $filesize\ /\ 4096\ bytes$ iterations, so a portion of our measured results are just the loop overhead, we believe the software overhead should grow as the size of the file increases. We also know that there is a ~12ns delay for taking time measurements, therefore this overhead will be included in each measurement of File read time.

We expect to see that as we increase the size of the file we are reading, the sequential average per-block reading time should only be increasing by a small amount whereas the random average per-block reading time should increase much more as we get towards the larger file sizes. In terms of time (ns), we performed some initial testing and saw that the average-per block time (ns) to read 1GB of file data sequentially was ~2000 ns and randomly was ~2500 ns. Based on these results we predict that the difference of 500ns between the two times will compound as the number of GBs being read increases.

Results:



Analysis:
From our results, we can conclude that the sequential average per-block read time is relatively constant as we increase in file size. However, it is interesting to see that initially the random average per-block read time is lower than the sequential read time up to 1GB. Then at around 8GB, we see a large divergence between the read times where it takes much longer for random read operations. We already know that caching should not affect the read times, therefore this means there is a difference between how fast those data blocks are accessed and read. From our interpretation of our graph, we believe that sequential access is better because of how modern disks optimize the layout of the data on disk. A discussion of what "sequential" means follows in the next paragraph.

We must explain the faults in this experiment. We are using MacOS to execute our experiments, and after thorough testing we were unable to effectively turn off or limit the file buffer cache. We first attempted to use the flag *F_NOCACHE* when opening the file, but we saw that this actually had no impact. We then tried to use shell commands such as *ulimit* to try to limit the FBC, but this was also unsuccessful. Therefore, for this experiment we believe that our results have been skewed because the FBC has not been turned off (it is not possible on MacOS).

In terms of "sequential" access of a file, we logically view it as going in consecutive blocks on disk. On the actual disk, this may not be true because even though we try to lay out blocks for a file system physically close to each other (using cylinder groups and disk caching to handle consecutive read misses) we may sometimes spill over to other cylinder groups and therefore potentially have more latency. However, that is not the common case. Therefore, sequential reads can take advantage of the consecutive layout of blocks on disk whereas random reads do not. We also know that modern disks can cache large amounts of consecutive data blocks that exist after the current data block being read. This optimization would allow for sequential reads to not have to truly "read from disk" as they will use the disk cache, but random reads may have to. We understand that we are using an SSD that does not have spinning disks or multiple heads however the argument of data block locality on disk and spilling over to other parts of disk still holds. The SSD will do its best to cluster similar data together, but at some point this will no longer be possible.

We are unsure as to why the major differentiation point between the two read times is at 8GB, but we believe this is due to the caching effects of not being able to turn off the FBC. We see a similar change in results right at 8GB in experiment *a* for file systems as well.

Note that we didn't use a standard deviation metric in this test as it was not useful in understanding our results.

c. Remote File Read Time:

Methodology:

To perform the following test, we used the UCSD ieng6 server as our "local machine" because for these machines the home directory is mounted over NFS, so accessing a file under the home directory will be a remote file access.

Since the account we used to access the ieng6 server had limited disk quota, we determined that it was valid to use a smaller file (0.5GB = 512MB) than what we used for our local file read time. Since we were now running on a Linux machine (CentOS Linux 7), we were able to use the *O_DIRECT* flag to explicitly read directly from the disk when performing I/Os and avoid caching effects. Note when opening the file, we specified the *O_DIRECT* and the *O_RDONLY* flags. We then malloc a buffer that is the size of a block (4096 bytes) in the file system on our machine (couldn't confirm this as all commands to get this information on ieng6 required sudo privileges).

Now we begin our file reading tests. For each file size, we increment in sizes of 16MB (16MB, 32MB, ..., 512MB). We perform both sequential reads and random reads. Note that because we are performing these I/Os directly from the disk, we do not need to worry about cache effects for reading the same data in and more every new file size or for reading the same file twice (sequentially then random). Once we receive the total time taken to perform the complete I/O for each file size, we store the results in a results file.

The sequential read operation is performed as follows: seek to the first byte of the file, read the file sequentially in 4096 byte chunks (time how long it takes to read each chunk from the file) until the complete file is read, and then return the average per-block read time.

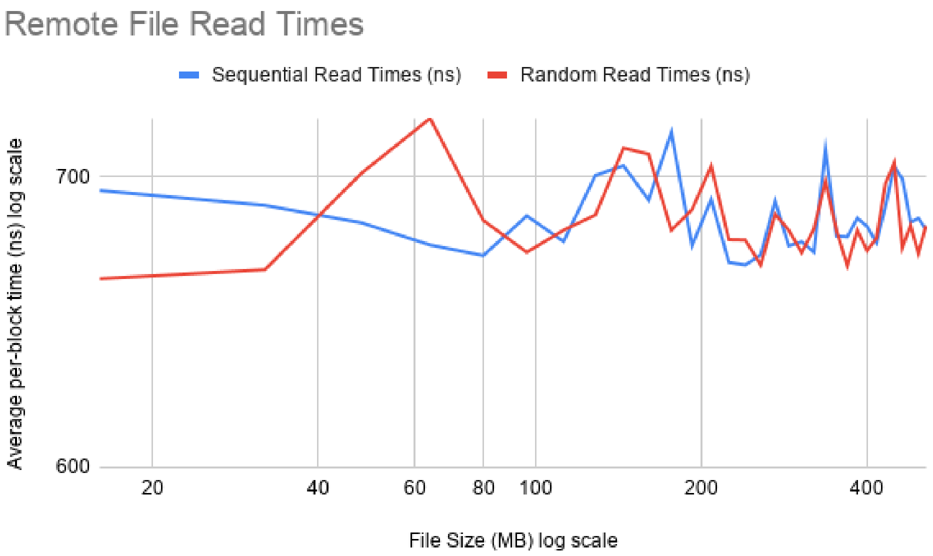
The random read operation is performed as follows: in a loop, select a random offset in the file that is not within 4096 bytes of the start or end of the file, seek to that byte offset the file, read 4096 bytes from that offset (time how long it takes to read the chunk from the file), repeat this process until we have read the specified file size number of bytes, and then return the average per-block read time.

Prediction:
Base Hardware: We are interacting with the disk in this experiment. According to our research, accessing the disk will require 1-3 ms. We also must take into account the disk transfer rate when we read from the disk as this will dictate how fast we can perform our reads. The network interface cards of both the ieng6 client server and the ieng6 NFS server are used to receive and transmit packets of file data across the network. Both of these network cards have a max bandwidth at which they can transmit data to and from the network. Therefore, the performance of these cards could limit the performance of the remote file data transfer. We were unable to find the actual maximum bandwidth values for the servers. In this test, there is also a network penalty involved since we must now wrap packets with the correct TCP/IP protocol, encrypt them, and transmit them through the network interface card. The same operations must then be performed in reverse to receive the data from the network. We believe the ping command and our results from Network Operation A are good estimates for this network penalty. We performed the ping command between the two ieng6 servers to gather this estimate and it was 0.388 ms. We also know that there is a ~12ns delay for taking time measurements, therefore this overhead will be included in each measurement of File read time.

Software Overhead: When performing a disk read we must execute the *read* syscall, transfer control to the kernel to handle the disk I/O, and then copying that data into main memory for the process to access. This is all done in software and we know that syscalls can take ~4000 ns which is relatively large. We also will see an increase in the number of instructions executed relative to how many times we iterate in our loop to read by block size. The loop will incur penalties in our time as well as the timing mechanisms we use. Another software overhead that could be added here is the loop overhead measured in experiment 1a. ii (roughly $2\mu s$). However, the loop in that test ran for 1000 times, whereas this experiment will run for the number of blocks in the file. That is each sample will run for $filesize\ /\ 4096\ bytes$ iterations, so a portion of our measured results are just the loop overhead, we believe the software overhead should grow as the size of the file increases.

We predict that differences in the time it takes to read sequentially from the disk versus randomly will be dwarfed by the network penalty. We assume that the ieng6 servers have strong enough I/O bandwidth such that the rate at which we transfer data is bottlenecked by the bandwidth of the network. Therefore, we expect to see little variation in the difference between random versus sequential reads for each file size as well as larger read average per-block read times than the local file read experiment.

Results:



Analysis:

These results are slightly different than what we had predicted initially. Our initial prediction was that the random versus sequential read times would not vary due to the network penalty. We predicted that the network penalty would mask the difference in time and therefore the average per-block read time would be similar.

The same discussion in part b about how sequential reads are actually performed can be applied here as well.

However, what is interesting is that the overall average per-block read time is significantly lower than the overall average per-block read time for the local file read average per-block read time. It is difficult to reason about this for the reasons we talk about in the next paragraph.

We cannot truly compare the results of the local file read time and the remote file read time as there are multiple different factors for each test. One is run on Ubuntu and the other on MacOS. One is run with a 0.5GB file and the other is run with a 16GB file. One has caching turned off and the other was not able to turn it off.

Note that we didn’t use a standard deviation metric in this test as it was not useful in understanding our results.

d. Contention:

Methodology:

We conducted the contention experiment using a bash script and C file. We have one C file that will read the filename passed in from the command line and open it with the read only flag using the *open* system call. We then malloc a buffer that is the size of a block (4096 bytes) in the file system on our machine (used the command *diskutil info / | grep "Block Size"* to confirm this). Next we read 1GB from the file by doing the following: seek to the first byte in the file, record the start time, in a loop, read from the file in block size chunks until we have read the complete amount of data we want to read, and record the end time. Finally, we calculate the time taken to read 1GB from the file, and write that time to a results file.

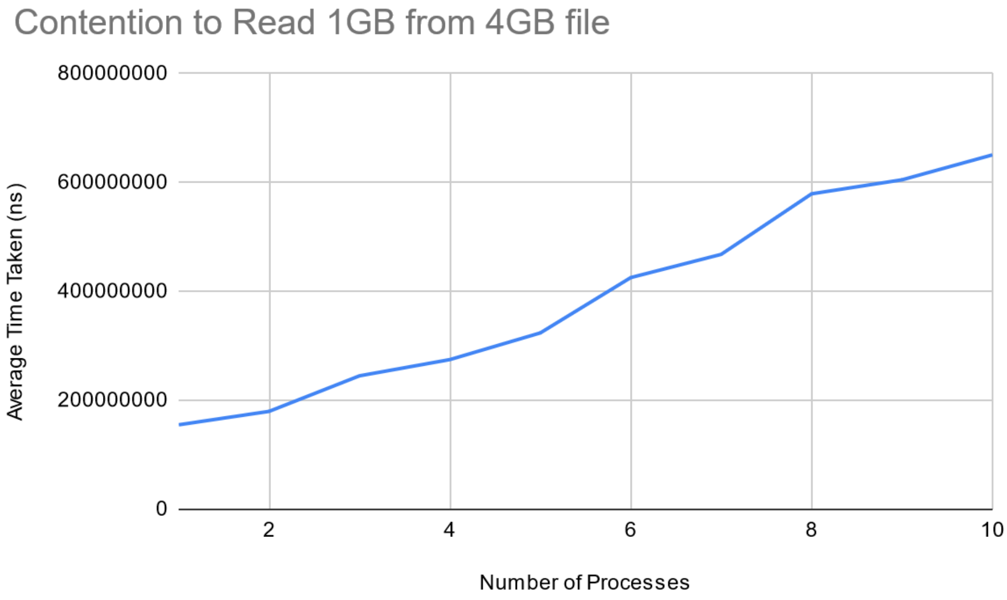
What was described in the paragraph above was for what one process will do. Next, we will explain how we conducted this test to see the average time taken to read the same amount of data while varying the number of processes performing the same program on different input data files in parallel. To run the experiment, we use a bash script that will loop through the number of parallel processes we want to test, and then create and execute a command that follows this template: *`./a.out file1 & ./a.out file2 & ... & ./a.out fileN`*. Remember that each process running will write it’s results to the same results file as all other processes however it will append its time taken to this file and therefore preserve any previous data in the file. It is interesting to note that as we increase the number of processes, we dynamically create new copies of the same file to pass into each *./a.out* execution. The reason for this is to avoid any caching effects of the disk since we are not able to perform raw disk I/Os directly from our C file. The flag F_NOCACHE was determined to be not working to read a file from disk as our results drastically changed running this experiment with the dynamic copies of the same file rather than the same file for all processes.

Prediction:
Base Hardware: We are interacting with the disk in this experiment. According to our research, accessing the disk will require 1-3 ms. We also must take into account the disk transfer rate when we page fault since this will affect how fast we can read data from the disk. The page fault service time is described further in the Software Overhead section. It is also important to think about parallelism in this case and if the base hardware is able to parallelize these processes on different cores. Therefore, this could actually improve the performance of the base hardware even though the number of parallel processes is increasing. However, according to the CSE221 lecture we learned that disk accesses must be serialized and this serialization happens at the hardware level. Therefore parallelism may not have a large impact on performance in this case,

Software Overhead: When we page fault, there will be extra overhead that we measure because we are now performing a disk read. When performing a disk read we must execute the *read* syscall, transfer control to the kernel to handle the disk I/O, and then copying that data into main memory for the process to access. This is all done in software and we know that syscalls can take ~4000 ns which is relatively large. From our memory operation C the time it takes for software to service a page fault is 36 ms, therefore the overhead for reading will be very large when we page fault. In addition to these overheads, we also have software overhead for context switches. There are many processes running at the same time and the scheduler will do it's best to give each process its' fair share of CPU time but that involves process context switch overhead which we measured in experiment 1e. ii which was roughly $25\mu s$. We cannot predict exactly how many context switches happen, but we know they will happen more often as we add more and more processes. We also know that there is a ~12ns delay for taking time measurements, therefore this overhead will be included in each measurement of this experiment.

We predict that as the number of parallel processes increases, we will have less effective use of the disk. Therefore, we predict the average time taken for I/O of the same amount of data from different copies of the same file across all processes will also be increasing.

Results:



Analysis:
Our results from our experiment are in line with our predicted results. As the number of parallel processes increases, the average time taken for I/O of the same amount of data from different copies of the same file across all processes will also be increasing. In short, with all other factors held constant except for the number of parallel processes and the number of different copies of the same file being read, we see that there is a decrease in performance as the number of parallel processes increases.

To understand this, we can look at the structure of the disk. The disk has a certain bandwidth at which it can transfer data from the disk into memory. When we have one process executing, that process will be able to use almost all of the disk bandwidth to perform its reads since the disk is not being used by any other process. However, as we increase the number of parallel processes reading from the disk, the disk will need to service each of the processes causing the disk bandwidth to be divided amongst each process (*n* processes means each process gets a disk bandwidth of *bandwidth/n*). Now each process will be reading in the same amount of data as before, but cannot transfer it as fast as if it was running by itself. This results in each process taking longer to complete its I/O and therefore increasing the time taken.

Note that we didn’t use a standard deviation metric in this test as it was not useful in understanding our results.

Final Results Table

Operation	Base Hardware Performance	Estimated Software Overhead	Predicted Time	Measured Time (Average)	Measured Time (Median)
1a (i)	N/A	15 ns ~ 62.5 ns	15 ns ~ 62.5 ns	12.032 ns	10.800 ns
1a (ii)	N/A	1500 ns	1500 ns	1983.2096 ns	1878.0000 ns
1b (0 args)	N/A	12 ns	12 ns	12.425120 ns	10.800000 ns
1b (1 args)	N/A	12 ns	12 ns	19.939520 ns	12.000000 ns
1b (2 args)	N/A	12 ns	12 ns	12.000000 ns	12.353360 ns
1b (3 args)	N/A	12 ns	12 ns	18.144160 ns	12.000000 ns
1b (4 args)	N/A	12 ns	12 ns	15.401760 ns	14.400000 ns
1b (5 args)	N/A	12 ns	12 ns	12.933760 ns	12.800000 ns
1b (6 args)	N/A	12 ns	12 ns	12.738640 ns	12.000000 ns
1b (7 args)	N/A	12 ns	12 ns	17.823760 ns	14.800000 ns
1c (*getpid bash)*	N/A	~200 ns	~200 ns	20.140000 ns	12.000000 ns
1c (*getpid c)*	N/A	~200 ns	~200 ns	14080.99639 ns	11831.2 ns
1c (*kill*)	N/A	~500 ns	~500 ns	4395.408800 ns	3658.80000 ns
1d (i)	N/A	10000 ns	10000 ns	11754.554399 ns	10910.9668 ns
1d (ii)	N/A	500000 ~ 1000000 ns	500000 ~ 1000000 ns	174440.2498 ns	149926.0 ns
1e (i)	N/A	100 ~ 3000 ns	100 ~ 3000 ns	6996.019 ns	7159.6 ns
1e (ii)	N/A	~5000 ns	~5000 ns	25929.597447 ns	21828.4 ns
2a (L1)	1.6 ns	12 ns			
2a (L2)	4 ns	12 ns			

2a (L3)	16 - 30 ns	12 ns	See results		
2a (DRAM)	60 ns	12 ns			
2b (Read) 16 loop unrolls	34.128 GB/sec	2 μs	34.128 GB/sec	15.892250 GB/sec	16.411738 GB/sec
2b (Read) 32 loop unrolls	34.128 GB/sec	2 μs	34.128 GB/sec	40.053863 GB/sec	40.866384 GB/sec
2b (Read) 64 loop unrolls	34.128 GB/sec	2 μs	34.128 GB/sec	103.478451 GB/sec	108.946651 GB/sec
2b (Write) 16 loop unrolls	34.128 GB/sec	2 μs	34.128 GB/sec	19.183353 GB/sec	19.50367 GB/sec
2b (Write) 32 loop unrolls	34.128 GB/sec	2 μs	34.128 GB/sec	52.3026476 GB/sec	54.46223 GB/sec
2b (Write) 64 loop unrolls	34.128 GB/sec	2 μs	34.128 GB/sec	119.89727 GB/sec	124.8808 GB/sec
2c	~10 μs	5-8 ms + ~12ns	~ 9 ms	36.76709392 ms	30.8491800 ms
3a (Local)	N/A	5000 ns + 12 ns = 5012 ns	103000 ns	221649.92 ns	193440.8 ns
3a (Remote)	46. 95012 *ns*	5000 ns + 12 ns = 5012 ns	106464000 ns	86696686.912 ns	85589230.4 ns
3b (Local) 0.5 GB	N/A	12ns	3827.2 MB/sec	411.60003 MB/sec	385.55867 MB/sec
3b (Local) 1 GB	N/A	12ns	3827.2 MB/sec	5954.34948 MB/sec	844.90041 MB/sec
3b (Local) 2 GB	N/A	12ns	3827.2 MB/sec	13896.21993 MB/sec	2538.49017 MB/sec

3b (Local) 4 GB	N/A	12ns	3827.2 MB/sec	24019.61066 MB/sec	4785.50701 MB/sec
3b (Local) 6 GB	N/A	12ns	3827.2 MB/sec	17498.04589 MB/sec	9125.13749 MB/sec
3b (Local) 8 GB	N/A	12ns	3827.2 MB/sec	108529.42285 MB/sec	7125.93147 MB/sec
3b (Local) 10 GB	N/A	12ns	3827.2 MB/sec	15418.52693 MB/sec	9313.70769 MB/sec
3b (Remote) 32MB	24 ms	(*Size* / 256) x 86ms	3.9 MB/sec	38.27136 MB/sec	37.99042 MB/sec
3b (Remote) 64MB	47.692 ms	(*Size* / 256) x 86ms	3.9 MB/sec	334.88487 MB/sec	106.98267 MB/sec
3b (Remote) 128MB	98.46 ms	(*Size* / 256) x 86ms	3.9 MB/sec	158.87030 MB/sec	110.47535 MB/sec
3b (Remote) 256MB	196.923 ms	(*Size* / 256) x 86ms	3.9 MB/sec	41.36245 MB/sec	296.94640 MB/sec
3b (Remote) 512MB	393.8461 ms	(*Size* / 256) x 86ms	3.9 MB/sec	40775.47702 MB/sec	662.82791 MB/sec
3b (Remote) 1024MB	787.69 ms	(*Size* / 256) x 86ms	3.9 MB/sec	5268.90420 MB/sec	1710.22834 MB/sec
3b (Remote) 2048MB	1.57 s	(*Size* / 256) x 86ms	3.9 MB/sec	2962.65423 MB/sec	2201.57866 MB/sec
3c (Local) Setup	N/A	5000 ns + 12 ns = 5012 ns	108012 ns	573685.328000 ns	343928.800000 ns
3c (Local) Teardown	N/A	5000 ns + 12 ns = 5012 ns	On the order of 10,000 ns, less than time to set up	75729.408000 ns	51800.800000 ns
3c (Remote) Setup	~47 *ns*	5000 ns + 12 ns = 5012 ns	106469012 ns	95011011.536 ns	93587943.6 ns

3c (Remote) Teardown	N/A	5000 ns + 12 ns = 5012 ns	On the order of 10,000 ns, less than time to set up	136107.120 ns	105210.400 ns
4a	1-3 ms + 12 ns	36 ms + 4000 ns + 12 ns	Linear growth followed by a spike and a dropoff after page fault(s).	See results	
4b	1-3 ms	*(Filesize /* 4096B) x 2µ*s* + 4000 ns + 12 ns	Slow linear growth for sequential access, but exponential growth for random access.	See results	
4c	1-3 ms + 0.388 ms + 12 ns	*(Filesize /* 4096B) x 2µ*s* + 4000 ns + 12 ns	Similar growth rates due to the network penalty.	See results	
4d	1-3 ms	36ms + 25µ*s* + 4000 ns + 12 ns	Linear growth as the number of competing processes increases.	See results	