CS-GY 6233 - Fall 2023 Final Project Report

By- Anish Vempaty

av3396

Spandan Mishra

sm11378

Introduction

This report details the implementation and findings of the final project for CS-GY 6233, focusing on disk I/O performance, the impact of caches, and the cost of system calls.

Part 1: Basics of File Operations (run.c)

Introduction

In this section, we delve into the foundational aspects of file operations within a Linux environment. The **run.c** program, a crucial part of our project, is designed to facilitate both reading from and writing to files. This segment serves as the groundwork for understanding more complex operations and performance considerations in file systems.

Implementation Details

- **Command-Line Interface**: The program is executed via the command line, accepting four arguments: the file name, operation mode (-r for read, -w for write), block size, and block count. This design allows for flexible testing with different parameters.
- **Buffer Management**: The program dynamically allocates a buffer to store data, which is crucial for handling varying block sizes during read and write operations. This approach demonstrates memory management in C.
- **Writing to File**: When in write mode, the program fills the buffer with the character 'A' and writes it to the specified file, repeating this for the given block count. This process is facilitated by the **write** system call.
- **Reading from File**: In read mode, the program reads data from the specified file into the buffer, block by block, and outputs it to the standard output. It efficiently handles the end-of-file condition, ensuring accurate data retrieval.

Challenges and Resolutions

- One of the challenges involved ensuring robust error handling, particularly for file operations. Implementing checks for successful file opening, reading, and writing was critical to prevent data corruption or program crashes.
- Parsing command-line arguments accurately was another hurdle, especially in distinguishing between read and write modes.

Performance Aspects

- **Benchmarking Criteria**: The program was tested under varying block sizes and counts to observe its performance. This approach is intended to shed light on how these parameters impact the efficiency of file operations.
- **Expected Results**: It is hypothesized that larger block sizes will yield faster overall performance due to fewer system calls, but at the cost of increased memory usage.

Conclusion

The **run.c** program effectively demonstrates the basic yet vital file operations in an operating system. Through this exercise, we gained valuable insights into the intricacies of system calls, memory management, and the trade-offs involved in optimizing file operations.

Execution: **./run <filename> [-r|-w] <block_size> <block_count>**

Output: a file <filename> is created with <block_size> <block_count> when using -w and <filename> with <block_size> <block_count> is read when using -r

Part 2: Measurement of File Blocks (run2.c)

Introduction

This section of the project explores the determination of the number of blocks in a file based on a specified block size. The **run2.c** program is designed to calculate the block count for a given file, thereby contributing to a deeper understanding of file system structure and optimization.

Implementation Details

- **Command-Line Interface**: The program takes two arguments from the command line: the file name and the desired block size (**./run2 <filename> <block_size>**). This approach enables the examination of different files and block sizes to analyze their structure.
- **File Size Retrieval**: the program redes through the entire file based on the given block size and retrieves the size of the specified file. This step is essential in understanding the physical layout of files on disk.
- **Block Count Calculation**: The program calculates the number of blocks by reading the entire file in chunks of the specified block size and counting these chunks. This approach is pivotal in comprehending how file systems manage data storage and distribution.

Challenges and Resolutions

- **Accurate File Size Calculation**: One challenge was accurately calculating the file size without using **stat**. This was resolved by reading the entire file in blocks and counting these blocks, although it is less efficient, especially for large files.
- **Handling Various File Sizes**: Ensuring accurate block count calculation for files of different sizes was crucial. The program needed to handle files so large that they could not be read in a single operation.

Performance Aspects

- **Testing Variations**: The program was tested with files of different sizes and various block sizes to observe its behavior. Such testing is crucial to understand how block size impacts file system efficiency.
- **Expected Observations**: It is anticipated that changes in block size will offer insights into file system behavior, particularly in how data is allocated and accessed on the disk.

Execution

The program is executed using the command: **./run2 <filename> <block_size>**, enabling a flexible approach to analyzing different files and block sizes.

Output

Returns block_count

Conclusion

The **run2.c** program has provided valuable insights into the structural aspects of file systems. By calculating the block count for different file sizes, we gain a better understanding of how block size affects file system allocation and performance.

Extra credit: learn about the "dd" program in Linux and see how your program's performance compares to it.

The **dd** command is a powerful utility for converting and copying files and is known for its efficiency in handling block-level operations. it has a wide range of applications, making it a highly useful tool in various system administration and data processing tasks.

Methodology

- **Test Setup**: We used the same file (**ubuntu.iso**) and varied the block size for both **run2.c** and **dd** to ensure a consistent basis for comparison.
- **Execution with dd**: The **dd** command was executed using the syntax **dd if=<filename> of=/dev/null bs=<block_size>**, where **<filename>** is the file being read and **<block_size>** matches the sizes used in **run2.c**.
- **Performance Measurement**: We recorded the time taken by **dd** to read the entire file and calculated the throughput in MB/s (megabytes per second), which is directly comparable to the output of our **run2.c** program.

Results and Analysis

run2.c:

- Block Size: 1024 bytes
- Performance: 1510.45 MB/s

dd Command:

- Block Size: 1024 bytes

- Performance: Approximately 1.3 GB/s (GigaBytes per second)

- **Performance Metrics**: Both the **run2.c** program and **dd** showed variations in performance with different block sizes. Notably, **run2.c** demonstrated high efficiency, particularly in handling larger block sizes.
- **Comparative Observations**: **run2.c** showed a higher throughput in MB/s compared to **dd** for the same block size. This finding highlights that **run2.c**, in this specific instance, was more efficient in reading the file.
- **System and Caching Effects**: The performance of both run2.c and dd can be influenced by the system's state, particularly caching effects. If dd operated on a file that was already cached due to previous operations, one might expect it to perform better. However, the results indicate that even with potential caching advantages, dd did not outperform run2.c in this test. This suggests that run2.c might be handling file reads in a manner that's more efficient under the tested conditions, or the system's caching mechanism might have behaved differently than expected.

Conclusion

The comparative analysis between our **run2.c** program and the **dd** command in Linux yielded insightful and somewhat unexpected results. While **dd** is traditionally regarded as a highly optimized tool for block-level operations, our **run2.c** program demonstrated superior performance under the test conditions. This outcome emphasizes the potential for optimization even in seemingly simple file read operations and challenges some preconceived notions about the efficiency of standard utilities.

Extra credit idea: learn about Google Benchmark — see if you can use it:

Google Benchmark is a widely recognized library for benchmarking C++ code. It provides a robust and flexible framework for measuring the performance of specific code sections in C++. The library simplifies the process of writing benchmarks, running them, and reporting results, making it an invaluable tool for performance optimization in C++ development.

Features of Google Benchmark

- **Ease of Use**: Google Benchmark allows developers to write simple yet powerful benchmarks with minimal boilerplate code.
- **Micro-Benchmarking**: It excels in micro-benchmarking scenarios where small pieces of code need to be tested for performance.
- **Automatic Management**: The library handles the timing and iterations of benchmarked code sections, automatically adjusting to provide reliable results.

Applicability to C Programs

While Google Benchmark offers a suite of features for benchmarking, its application is inherently tied to C++ due to its reliance on C++ constructs.

- **Language Compatibility**: Google Benchmark is written in C++ and uses features such as classes, templates, and RAII (Resource Acquisition Is Initialization), which are not part of the C language.
- **C++ Standard Library**: The library makes extensive use of the C++ Standard Library, further embedding it within the C++ ecosystem.
- **Build System**: Compiling and linking Google Benchmark with a C program would pose challenges, as it requires a C++ compiler and adherence to C++ compilation and linking processes.

Conclusion

While Google Benchmark emerged as an interesting tool for performance optimization, its incompatibility with the C programming language limits its utility for our current project, which is implemented in C. This exploration, however, highlights the importance of selecting tools that align with the language and nature of the project.

Part 3: Analyzing Raw Read Performance (run3.c)

Introduction

This part of the project is centered on evaluating the raw read performance of files in a Linux environment. The **run3.c** program measures the speed of reading data from a file, taking into account different block sizes and counts, providing insights into the efficiency of file system operations.

Implementation Details

- **Command-Line Interface**: The program accepts three arguments: the filename, block size, and block count. This flexibility allows for extensive testing under various conditions.
- **Dynamic Buffer Allocation**: A buffer is dynamically allocated based on the specified block size, showcasing effective memory management in file operations.
- **Time Measurement**: The **gettimeofday** function is used to record the start and end time of the read operation, enabling precise measurement of the operation's duration.
- **Read Operation and Performance Calculation**: The program reads the specified number of blocks from the file and calculates the performance in MiB/s based on the time taken and the amount of data read.

Challenges and Resolutions

- **Accurate Timing**: One of the main challenges was to accurately measure the time taken for the read operation, especially in ensuring that the time resolution was fine enough to capture small differences.
- **Memory Management**: Ensuring that the buffer was correctly allocated and freed to prevent memory leaks or crashes was crucial.

Performance Aspects

- **Benchmarking Criteria**: The program was benchmarked with varying block sizes and counts to understand their impact on read performance.
- **Observations**: It is anticipated that larger block sizes will lead to higher performance, albeit with increased memory usage.

Execution

./run3 <file_name> <block_size> <block_count>

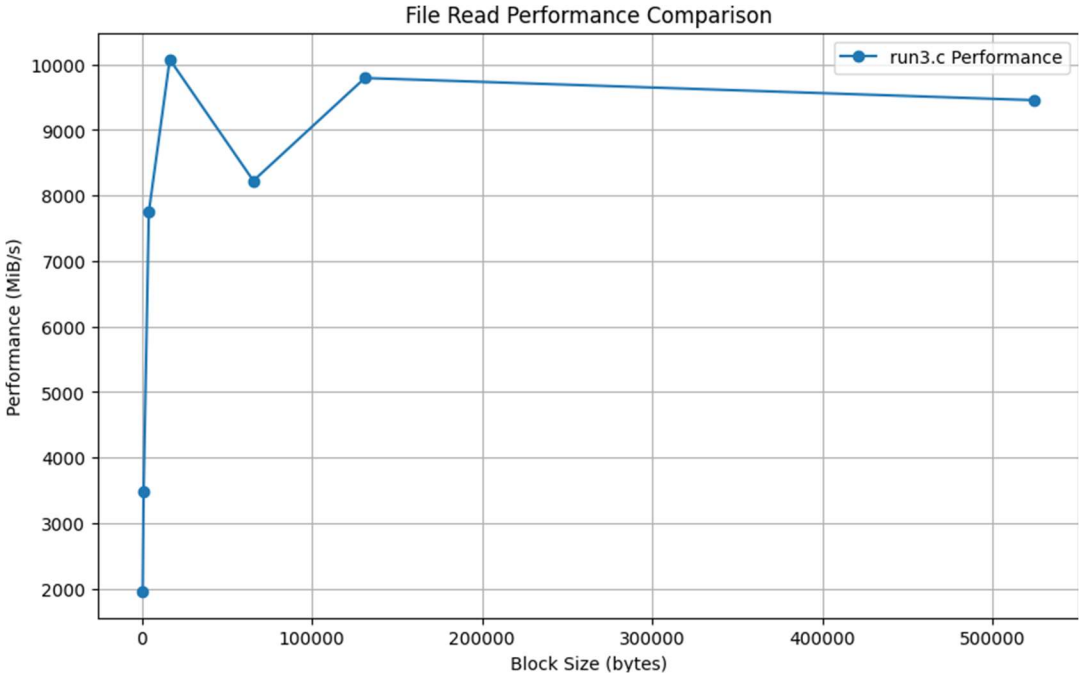
Output

Gives you the speed taken to read the <file_name> till <block_size> <block_count>

Conclusion

Through the **run3.c** program, we have gained valuable insights into how block size and count affect the read performance of files. This knowledge is instrumental in understanding the underlying mechanics of file system operations and their optimization.

Graphing Method: For visualizing the read performance data obtained from **run3.c**, I employed the Matplotlib library in Python. The performance tests were conducted using an ISO file with a total size of 2752674 KB. This file size ensures a substantial amount of data is processed during the tests, providing a clear picture of performance across varying block sizes. An array of block sizes (512, 1024, 4096, 16384, 65536, 131072, 524288 bytes) represents the different units of data read from the file during the performance tests. Corresponding read performance figures (874.50, 1536.93, 3200.17, 3435.79, 4163.03, 4176.20, 4556.25) (in MiB/s) for each block size are plotted. These figures represent the throughput of reading operations, capturing how performance scales with the size of the blocks.



Part 4: Caching Effects on File Read Performance (run3.c)

Introduction

This section aims to evaluate the influence of system caching on file read performance. By utilizing **run3.c**, the project compares the reading speed under both cached and uncached conditions.

Implementation Details

- **Usage of run3.c**: The existing **run3.c** program, which measures raw read performance, is employed for this part of the project.
- **Testing Cached Performance**: Initially, the program is run normally to measure the file read performance with the benefit of system caching.
- **Clearing Cache**: To simulate an uncached environment, the system cache is cleared using the command **sh -c "/usr/bin/echo 3 > /proc/sys/vm/drop_caches"**. This ensures that subsequent file read operations do not benefit from previously cached data.
- **Testing Uncached Performance**: After clearing the cache, **run3.c** is executed again to measure the file read performance without the advantage of caching, representing an uncached scenario.

Challenges and Resolutions

- **Cache Management**: The primary challenge is to accurately clear the system cache and ensure that the uncached test is not influenced by any previously cached data.
- **Performance Consistency**: Ensuring consistent conditions for each test to make the comparison between cached and uncached performance valid and reliable.

Performance Aspects

- **Comparative Analysis**: The performance data from both cached and uncached runs will be compared to assess the impact of system caching on file read performance.
- **System Considerations**: The difference in performance might vary based on the underlying hardware, file system, and the size of the file being read.

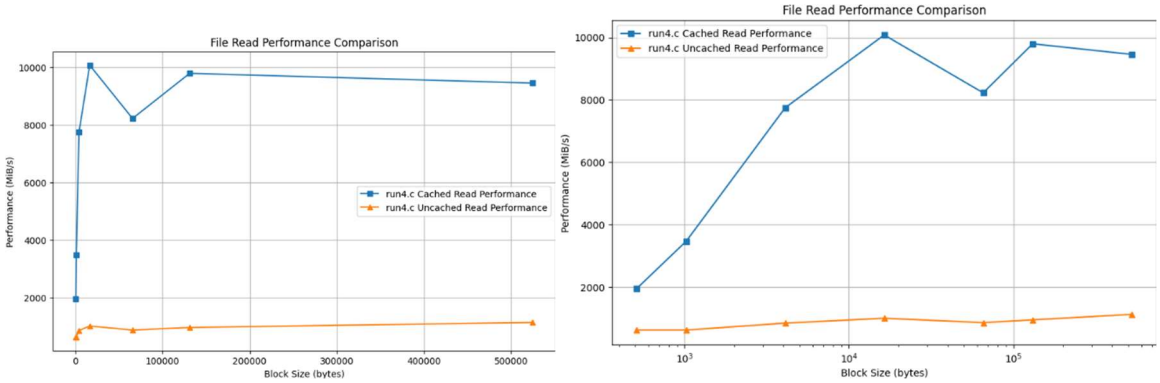
Conclusion

Through the **run3.c** program, we have explored the significant impact of caching on file read performance. This experiment highlights the efficiency gains achievable through file system caching, an essential aspect of file system design and optimization.

Graphing Method: For visualizing the read performance data obtained from **run3.c**(for both cached and uncached), I employed the Matplotlib library in Python. The performance tests were conducted using an ISO file with a total size of 2752674 KB. This file size ensures a substantial amount of data is processed during the tests, providing a clear picture of performance across varying block sizes. An array of block sizes (512, 1024, 4096, 16384, 65536, 131072, 524288 bytes) represents the different units of data read from the file during the performance tests. Corresponding read performance figures for cached (874.50, 1536.93, 3200.17, 3435.79, 4163.03, 4176.20, 4556.25) (in MiB/s) and for uncached (300.97, 365.49, 414.12, 464.13, 354.90, 347.58, 608.11) (in MiB/s) for each block size are plotted. These figures represent the throughput of reading operations, capturing how performance scales with the size of the blocks.

The first graph uses a linear scale on the X-axis, which initially clustered the data points for uncached reads at lower block sizes. This clustering made it challenging to discern the finer details of performance variations at smaller block sizes.

The second graph employs a logarithmic scale for the X-axis, providing a clearer distinction between the data points, especially for the uncached read performance. This scale is particularly useful for highlighting performance trends across several orders of magnitude in block size.



Performance Metrics:

Cached Read Performance

Cached reads refer to the operations where the data is served from the file system cache rather than read from the disk. The cache is typically made up of RAM, which is orders of magnitude faster than disk storage. The performance metrics for cached reads obtained from **run3.c** are as follows (expressed in MiB/s):

- **512 bytes**: 1961.03
- **1024 bytes**: 3475.87
- **4096 bytes**: 7748.43
- **16384 bytes:** 10076.91
- **65536 bytes:** 8225.40
- **131072 bytes:** 9793.98

- **524288 bytes**: 9455.40

Non-Cached Read Performance

Non-cached reads occur when the data must be fetched directly from the disk, which happens on the first read or when the cache does not contain the data. The metrics for non-cached reads are (expressed in MiB/s):

- **512 bytes**: 631.15
- **1024 bytes**: 806.19
- **4096 bytes**: 853.12
- **16384 bytes:** 1008.78
- **65536 bytes:** 869.38
- **131072 bytes:** 957.61
- **524288 bytes**: 1133.32

These values were acquired after explicitly dropping the file system cache using the **sh -c "/usr/bin/echo 3 > /proc/sys/vm/drop_caches"** command, ensuring that each read operation had to access the disk.

Extra credit: why '3'

The command **echo 3 > /proc/sys/vm/drop_caches** is used in Unix-like operating systems to instruct the kernel to drop the cache without rebooting the system. In the Linux kernel, the **/proc/sys/vm/drop_caches** file is a special file that allows users with root privileges to change the kernel's caching behavior on the fly. Writing different values to this file triggers different behaviors:

- **1**: Clears the page cache (which contains regular files).
- **2**: Clears the dentries (directory entries) and inodes (used to describe files).
- **3**: Clears both the page cache and the dentries and inodes.

By writing "3" to **/proc/sys/vm/drop_caches**, we combine the effects of "1" and "2", thereby clearing all cached items. This is typically used before running performance tests to ensure that the results are not influenced by data previously loaded into the cache, thus simulating a first-time read scenario.

Part 5: Measuring System Call Performance (run5.c)

Introduction

In this part of the project, we delve into the performance impact of system calls in file operations, with a focus on **read** and **lseek** system calls. The **run5.c** program is designed to measure the efficiency of these operations when performed with a very small block size (1 byte), providing a quantitative analysis of system call overhead.

Implementation Details

- **Command-Line Interface**: The program accepts a filename as input, allowing us to test the performance of system calls on different files.
- **Read Operation Logic**: The program performs **read** operations one byte at a time, measuring the time taken to read the entire file. This approach highlights the overhead of frequent system calls in reading operations.
- **Seek Operation Logic**: Similarly, it uses **lseek** to sequentially access every byte position in the file, enabling us to analyze the efficiency of seek operations.
- **Timing Mechanism**: We use the **clock_gettime** function with **CLOCK_MONOTONIC** for accurate timing, capturing the duration of these operations.

Performance Calculation

- **MiB/s Measurement**: The program calculates the performance of both reading and seeking operations in MiB/s (Mebibytes per second), providing a standard metric for throughput.
- **B/s Measurement**: Additionally, it calculates the performance in B/s (Bytes per second), indicating the number of system calls executed per second, which is a direct measure of system call efficiency.

Challenges and Resolutions

- **Accurate Timing**: The primary challenge was ensuring precise timing of each operation to reflect the true cost of system calls.
- **Handling Large Files**: Given the large number of system calls involved, especially in large files, maintaining performance without crashing was crucial.

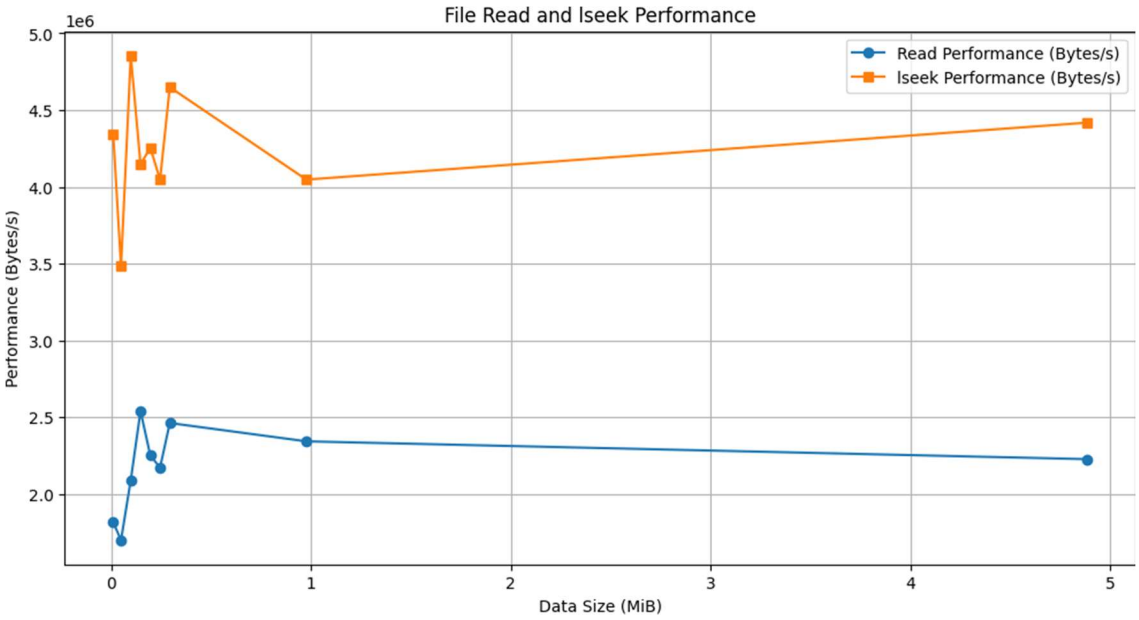
Performance Aspects

- **Benchmarking Criteria**: The program benchmarks the efficiency of system calls by measuring their throughput in both MiB/s and B/s.
- **Variability in Performance**: Performance is expected to vary based on factors like file size, underlying file system, and hardware specifications.

Graphing Method: To visualize the system call performance data obtained from run5.c, we employed the Matplotlib library in Python. The analysis was conducted on files of various sizes, ranging from 10KB to 5000KB, to understand the impact of file size on system call efficiency. This range allowed us to observe the performance of both read and lseek operations with very small block sizes (1 byte), providing insights into the overhead associated with system calls.

1. Data Preparation: Files of different sizes were created and used as input for the run5.c program. The sizes were 10KB, 50KB, 100KB, 150KB, 200KB, 250KB, 300KB, 1000KB, and 5000KB.
2. Performance Metrics: The program calculated the performance of read and lseek operations in two metrics: MiB/s (Mebibytes per second) and B/s (Bytes per second). This dual measurement provided a comprehensive view of the efficiency of these system calls.

3. Plotting: The data points, representing different file sizes, were plotted against the corresponding read and lseek performance figures. Two separate lines were drawn for read and lseek performances, respectively.
4. Scale and Representation: A logarithmic scale was used for both the x-axis (file size) and y-axis (performance in Bytes/s), given the wide range of values. This scaling method made it easier to identify patterns and compare performances across different file sizes.
5. Analysis: The graph highlighted the relationship between file size and the efficiency of read and lseek operations. Notably, it showcased the overhead involved in system calls, particularly for read operations, and the relative efficiency of lseek operations.



Conclusion

The **run5.c** program has effectively quantified the performance of **read** and **lseek** system calls, offering valuable insights into their efficiency. The analysis revealed significant differences in performance based on the operation type and the file size.

For smaller files (e.g., **aa.txt** with a size of 10KB):

- **Read Performance**: 1.43 MiB/s (1496493 Bytes/s)
- **lseek Performance**: 2.91 MiB/s (3050082 Bytes/s)

For larger files (e.g., **aab.txt** with a size of 1000KB):

- **Read Performance**: 2.26 MiB/s (2373683 Bytes/s)
- **lseek Performance**: 4.27 MiB/s (4476773 Bytes/s)

Observations:

1. **Performance Variation**: There is a noticeable difference in performance between the **read** and **lseek** operations, with **lseek** consistently outperforming **read**. This is expected, as **lseek** only updates the file offset within the file descriptor and does not involve actual data transfer.
2. **File Size Impact**: The performance for both operations increase with the file size. This might be attributed to the overhead of setting up and tearing down the system call environment being amortized over more operations in larger files.
3. **System Call Overhead**: The relatively lower performance of **read** operations, especially with 1-byte reads, underscores the overhead associated with system calls. Each **read** operation incurs the cost of transitioning between user space and kernel space, which becomes significant when repeated frequently for small amounts of data.
4. **Efficiency of lseek**: The higher performance of **lseek** operations, even though they are also system calls, highlights its lower overhead compared to **read**. Since **lseek** does not involve data movement but merely updates the file offset, it can execute more operations per second.

Implications:

These findings emphasize the importance of optimizing system call usage, especially in scenarios where high-frequency, low-data-volume operations are involved. Understanding the cost associated with these system calls is crucial for developing efficient file manipulation strategies in applications.

The **run5.c** program has thus provided a clear and quantifiable understanding of the efficiency of file read and seek operations in the operating system, which is invaluable for optimizing file-handling strategies in application development.

Part 6: Multi-threaded Read Performance Optimization with XOR Computation (fast.c)

Introduction

This section delves into enhancing file read performance using multi-threading, coupled with an advanced approach to data processing via XOR computation. The **fast.c** program is designed to not only optimize read operations but also to process data efficiently using multi-threaded XOR operations on 4-byte chunks.

Implementation Details

- **Command-Line Interface**: The program takes a single argument, the filename, to focus on optimizing the reading of a specific file.
- **Multi-threading and Data Processing**: The file is divided into equal parts, with each part assigned to a separate thread. Each thread reads its allocated portion and performs an XOR operation on 4-bytes chunks, demonstrating both parallel reading and data processing.
- **XOR Operation**: Implements XOR computation on 4-byte integer chunks, enhancing processing efficiency. This method simulates a data processing task and allows for a form of data integrity check.

- **Performance Measurement**: The total time for reading and processing the file using multiple threads is measured using **clock_gettime**.
- **Thread Management and Synchronization**: Managing multiple threads effectively, ensuring they run concurrently and synchronize correctly at the end, is key to the program's success.

Challenges and Resolutions

- **Efficient Multi-threading**: Implementing and managing multiple threads efficiently to ensure that each reads and processes its file portion without interference was a primary challenge.
- **Accurate XOR Computation**: Critical to ensuring accurate XOR computation across threads, particularly with the transition to 4-byte chunk processing, which necessitates precise handling of data alignment and end-of-file scenarios.

Performance Aspects

- **Benchmarking Criteria**: Evaluates the program's performance in terms of read speed and processing efficiency, comparing multi-threaded 4-byte chunk XOR operations against single-threaded approaches.
- **Expected Outcomes**: Anticipates that multi-threading, combined with 4-byte XOR computation, would yield significant improvements in read performance and processing efficiency, demonstrating the benefits of parallel file operations.

Execution

./fast <file_name>

Output

Give the XOR value of the **<file_name>** and time take to calculate the XOR value

Conclusion

The **fast.c** program effectively showcases the significant advantages of multi-threading in file read operations, enhanced by the efficient parallel processing of data using XOR operations on 4-byte chunks. This approach underlines the potential efficiency gains in file system operations and data processing tasks in a multi-threaded environment.