A Course Based Project Report
On
PARALLEL FILE ENCRYPTOR

Submitted in partial fulfillment of requirement
for the completion of the
Operating Systems

II B.Tech Computer Science and Engineering
of
VNR VJIET

By

G. SRI VIDYA	**23071A05G1**
K. ADVAITH	**23071A05G5**
K. KARTHIKEYA	**23071A05G6**
K. NIKHITHA	**23071A05G9**
SRINIDHI	**24075A0517**

2023-2024

VALLURIPALLI NAGESWARA RAO
VIGNANA JYOTHI INSTITUTE OF ENGINEERING &TECHNOLOGY
(AUTONOMOUS INSTITUTE)
NAAC ACCREDITED WITH ‘A++’ GRADE
NBA Accreditation for B. Tech Programs
Vignana Jyothi Nagar, Bachupally, Nizampet (S.O), Hyderabad 500090
Phone no: 040-23042758/59/60, Fax: 040-23042761
Email:postbox@vnrvjiet.ac.in Website: www.vnrvjiet.ac.in

A Project Report On

PARALLEL FILE ENCRYPTOR

Submitted in partial fulfillment of requirement
for the completion of the
Operating Systems

2023-2024

B.Tech Computer Science and Engineering

Of

VNRVJIET

2023 – 2024

Under the Guidance

Of

DR. D N VASUNDHARA
Assistant Professor
Department of CSE

VNR VIGNANA JYOTHI INSTITUTE OF ENGINEERING & TECHNOLOGY

(AUTONOMUS INSTITUTE)

NAAC ACCREDITED WITH ‘A++’ GRADE

CERTIFICATE

This is to certify that the project entitled “PARALLEL FILE ENCRYPTOR” submitted in partial fulfillment for the course of Operating System being offered for the award of Batch (CSE-C) by VNRVJIET is a result of the bonafide work carried out by **23071A05G1, 23071A05G5, 23071A05G6, 23071A05G9 and 24075A0517** during the year **2023-2024**.

This has not been submitted for any other certificate or course.

.

Signature of Faculty **Signature of Head of the Department**

ACKNOWLEDGEMENT

An endeavor over a long period can be successful only with the advice and support of many well-wishers. We take this opportunity to express our gratitude and appreciation to all of them.

We wish to express our profound gratitude to our honorable **Principal Dr. C.D.Naidu** and **HOD Dr.V Baby**, **CSE Department**, **VNR Vignana Jyothi Institute of Engineering and Technology** for their constant and dedicated support towards our career moulding and development.

With great pleasure we express our gratitude to the internal guide **DR. DN VASUNDHARA, Assistant Professor, CSE department** for her timely help, constant guidance, cooperation, support and encouragement throughout this project as it has urged us to explore many new things.

Finally, we wish to express my deep sense of gratitude and sincere thanks to our parents, friends and all our well-wishers who have technically and non-technically contributed to the successful completion of this course-based project.

DECLARATION

We hereby declare that this Project Report titled **“PARALLEL FILE ENCRYPTOR”** submitted by us of Computer Science & Engineering in **VNR Vignana Jyothi Institute of Engineering and Technology** is a bonafide work undertaken by us and it is not submitted for any other certificate/Course or published any time before.

Name & Signature of the Students:

G. SRI VIDYA	**23071A05G1**
K. ADVAITH	**23071A05G5**
K. KARTHIKEYA	**23071A05G6**
K. NIKHITHA	**23071A05G9**
SRINIDHI	**24075A0517**

Date:

TABLE OF CONTENTS

ABSTRACT

In today’s digital-first world, data security is no longer just a feature—it is a necessity. As we continue to generate and store more data every day, from personal files on our devices to massive enterprise datasets in the cloud, the need to secure this information from unauthorized access becomes critical. Encryption is one of the most widely used and effective techniques to protect data by converting it into a form that cannot be easily understood without a decryption key.

However, a common drawback of encryption—especially when dealing with large files—is that it can be time-consuming. Traditional single-threaded approaches process files sequentially, often leading to performance bottlenecks, particularly when working with high volumes of data.
Our project proposes a solution to this problem through the implementation of a **Parallel File Encryptor in C++**.

By leveraging the power of multithreading, the encryptor divides a file into smaller chunks and processes them simultaneously across multiple CPU cores. This approach significantly reduces encryption time and utilizes modern hardware capabilities more efficiently. The choice of C++ as the programming language provides low-level control over system resources and enables high-performance computation, which is crucial for tasks like file encryption.

To demonstrate the concept, we have implemented a simple XOR-based encryption algorithm. This lightweight encryption logic allows us to focus more on demonstrating the benefits of parallelism. However, the design is highly modular, which means stronger cryptographic algorithms like AES or Blowfish can be integrated in future versions without rewriting the entire system.

This project is not just a demonstration of parallel computing—it is a practical tool that blends core concepts from operating systems, cryptography, and systems programming. It addresses a real-world problem with a scalable and efficient solution. By enabling faster file encryption through parallelism, it opens the door to more secure systems that don’t compromise on performance, especially in enterprise environments, cloud platforms, and personal devices handling large files daily.

INTRODUCTION

As our dependency on digital technology continues to grow, so does our responsibility to protect the data we interact with. Whether it's personal media, confidential documents, or sensitive corporate records, the value of secure data handling has never been higher. Unfortunately, with growing data sizes and increasing cyber threats, maintaining both security and speed has become a challenging balancing act.

Encryption, as a defense mechanism, transforms data into an unreadable format, ensuring that only authorized parties can access the original content using a specific decryption key. It is the foundation of modern digital security—used in everything from messaging apps and cloud storage to banking systems and government archives. Despite its strengths, one notable drawback of encryption—especially with large files—is the processing time.

Traditional encryption software often works in a sequential, single-threaded manner. This means the system processes the file one segment at a time, which can be painfully slow, particularly when dealing with gigabytes of data.

Our project, a **Parallel File Encryptor written in C++**, tackles this exact problem. The key innovation lies in utilizing multithreading to handle different parts of a file simultaneously. Most modern processors today come with multiple cores, capable of running several tasks in parallel. This project taps into that hardware advantage by dividing a file into chunks and encrypting each chunk using a separate thread. As a result, we see a noticeable improvement in speed, especially on multi-core systems.

The reason we chose C++ is because of its strong performance characteristics and direct access to system-level programming. It allows us to manage memory and threads efficiently while building a system that remains flexible and powerful. For encryption, we started with a basic XOR-based algorithm.

This allows us to validate the working of the parallel system without the overhead of complex cryptography. The overall design, however, is kept modular, meaning stronger encryption algorithms can be plugged in later as needed.

In short, this project is a practical exploration of how to make encryption faster without compromising security. It brings together concepts of file handling, multithreading, and cryptography to deliver a real-world solution. With this tool, users and developers can enjoy fast, secure, and scalable file encryption—exactly what is needed in the data-heavy age we live in.

METHODOLOGY

The design and development of the Parallel File Encryptor followed a clear, systematic methodology that allowed us to transform a traditional, sequential encryption process into a fast and efficient multithreaded application. Our approach focused on maintaining simplicity in the encryption logic while maximizing performance through effective thread management and safe file operations.

1. File Chunking and Preparation

- The first task was to break down the input file into smaller chunks. These chunks are designed to be approximately equal in size so that they can be distributed evenly among multiple threads.
- Depending on the total file size and the number of threads available (which is typically determined by the number of processor cores), we calculate how many chunks to create.
- This chunking process is vital because it forms the basis for parallel execution—each chunk can be processed independently.

2. Launching Threads for Parallel Processing

- Once the chunks are defined, we launch multiple threads using the C++ <thread> library.
- Each thread is assigned a specific chunk of the file and tasked with encrypting it. These threads run in parallel, which means several parts of the file are being processed at the same time.
- This is the heart of the performance gain in our project. On a multi-core CPU, this parallelism allows encryption to be completed much faster than with a single-threaded approach.

3. Applying XOR Encryption

- To keep things straightforward and lightweight during testing, we implemented a simple XOR-based encryption algorithm. This method takes each byte in a chunk and performs an XOR operation with a predefined key.
- The simplicity of XOR makes it an excellent choice for this prototype, as it is fast, reversible, and easy to implement.
- What makes XOR especially handy is that the same operation can be used for both encryption and decryption—simply running it twice with the same key restores the original data.

4. **Ensuring Thread Safety**
   - Multithreaded applications can become tricky when threads share memory or try to access the same resource. To avoid such issues, our design ensures that each thread works on its own isolated chunk of data.
   - There’s no overlapping memory access, so we avoid race conditions and the need for complex synchronization mechanisms like mutexes. However, we are careful with file reading and writing positions to ensure that threads don’t interfere with each other’s I/O operations.

5. **Writing the Output File**
   - Once encryption is complete, the encrypted chunks are written back to a new file (or overwrite the original, if chosen).
   - This step can be done in a sequential manner to maintain the correct order of data.
   - Alternatively, parallel writing can also be used with careful buffer management. Our current implementation focuses on preserving data integrity and keeping the file structure consistent for easy decryption.

6. **Testing and Performance Analysis**
   - Finally, we tested the application with files of varying sizes—from a few kilobytes to several gigabytes. We measured how the execution time changed when using different numbers of threads.
   - The results clearly demonstrated that parallel processing can reduce encryption time significantly, especially for larger files, validating the core idea of our project.

In conclusion, this methodology not only showcases the advantages of parallelism in encryption but also creates a foundation for building more advanced tools in the future. Whether used in secure backup systems, encrypted cloud storage, or personal file security, this approach proves that faster encryption is very achievable with the right design principles.

OBJECTIVES

The main objective of this project is to design and implement a **Parallel File Encryptor** in C++ that efficiently secures data while leveraging the capabilities of modern multi-core processors. The focus is on enhancing performance and scalability without compromising simplicity or reliability. The specific goals are outlined below:

1. **Improve File Encryption Speed Through Parallelism:**
   To significantly reduce the time taken to encrypt large files by utilizing multithreading, thereby allowing different parts of a file to be processed simultaneously across multiple CPU cores.

2. **Demonstrate Effective Use of Multithreading in C++:**
   To explore and apply C++ multithreading constructs (using <thread>, <mutex>, and other libraries where applicable) for parallel file operations, showcasing how concurrency can optimize performance in real-world applications.

3. **Implement a Lightweight Yet Functional Encryption Algorithm:**
   To develop and integrate a simple XOR-based encryption scheme that clearly demonstrates the functionality of parallel processing, while also laying the groundwork for stronger cryptographic algorithms in future iterations.

4. **Build a Modular and Scalable System:**
   To ensure the encryptor is built with modularity in mind, allowing the easy replacement or upgrade of components such as the encryption logic or threading model without rewriting the entire application.

5. **Ensure File Integrity and Thread Safety:**
   To design the system in such a way that each thread handles its own chunk of data without conflicts or data corruption, maintaining thread safety and file consistency throughout encryption and decryption processes.

6. **Provide a User-Friendly Command-Line Interface (CLI):**
   To offer a simple and intuitive user interface that allows users to easily specify files, encryption keys, number of threads, and output paths, making the tool practical for real-world usage.

In conclusion, the objectives of this project aim to combine secure file encryption with the power of parallel processing to enhance performance. By leveraging C++ multithreading, the system is designed to be efficient, scalable, and practical for real-world usage. These goals collectively contribute to a deeper understanding of concurrent programming and its applications in cybersecurity.

FLOW OF EXECUTION

The execution of the project is divided into two primary branches:

a) Child Process-based Execution (Multiprocessing):

1. The main process reads the input file and splits the content into chunks.
2. For each chunk, a child process is created using the fork() system call.
3. Each child process encrypts or decrypts its chunk independently.
4. The encrypted/decrypted data is passed back to the parent process via temporary files or pipes.
5. The parent process collects all the processed chunks and combines them into the final output file.

b) Multithreaded Execution (Shared Memory):

1. The main process divides the input file into memory segments.
2. POSIX threads (pthreads) are created, with each thread handling a segment.
3. Shared memory segments are used for efficient communication between threads.
4. Semaphores ensure synchronization to avoid race conditions.
5. Once all threads complete processing, the results are merged into the final file.

Both implementations follow a similar high-level flow:

File Input

↓

Task Division

↓

Parallel Execution

↓

Result Aggregation

↓

File Output

OUTPUT

Sample outputs for both implementations were captured and verified:

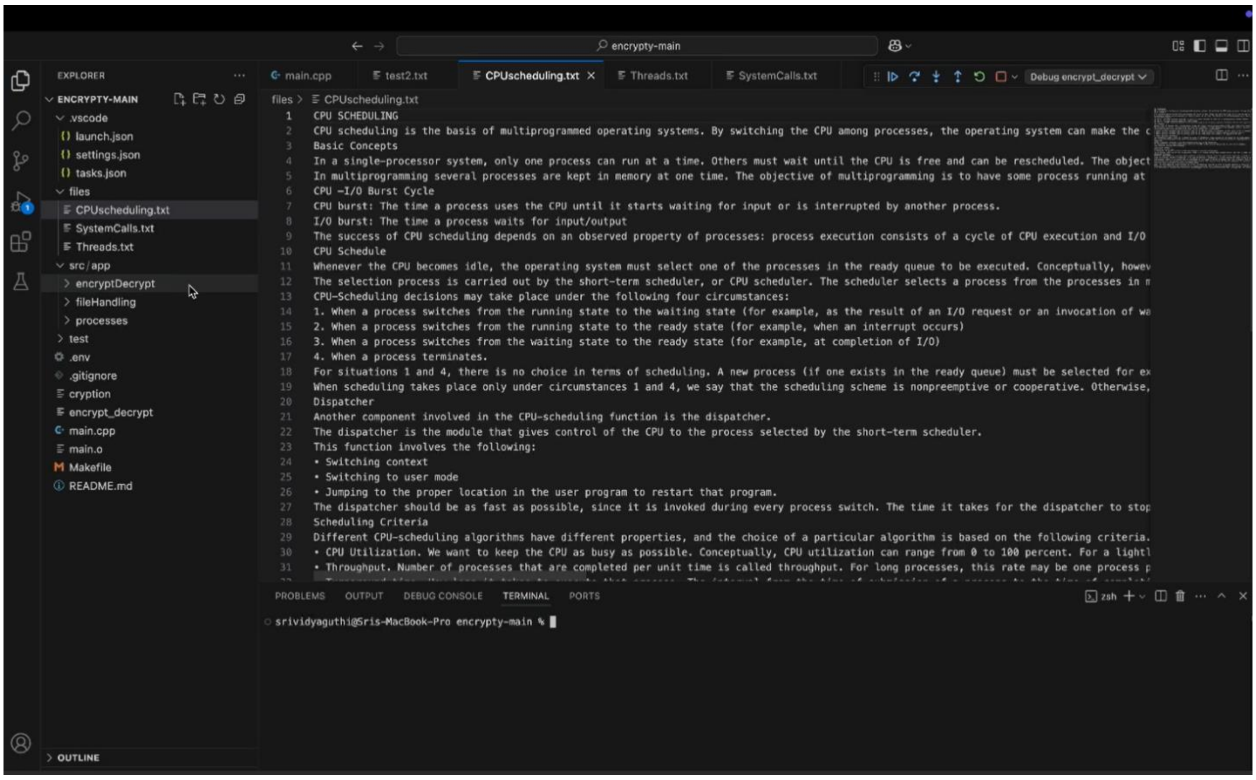
a) Encryption Output (Multiprocessing):

- Input: HelloWorld
- Encrypted Output (Example): KhoorZruog *(Caesar Cipher with shift 3)*

b) Decryption Output (Multithreading):

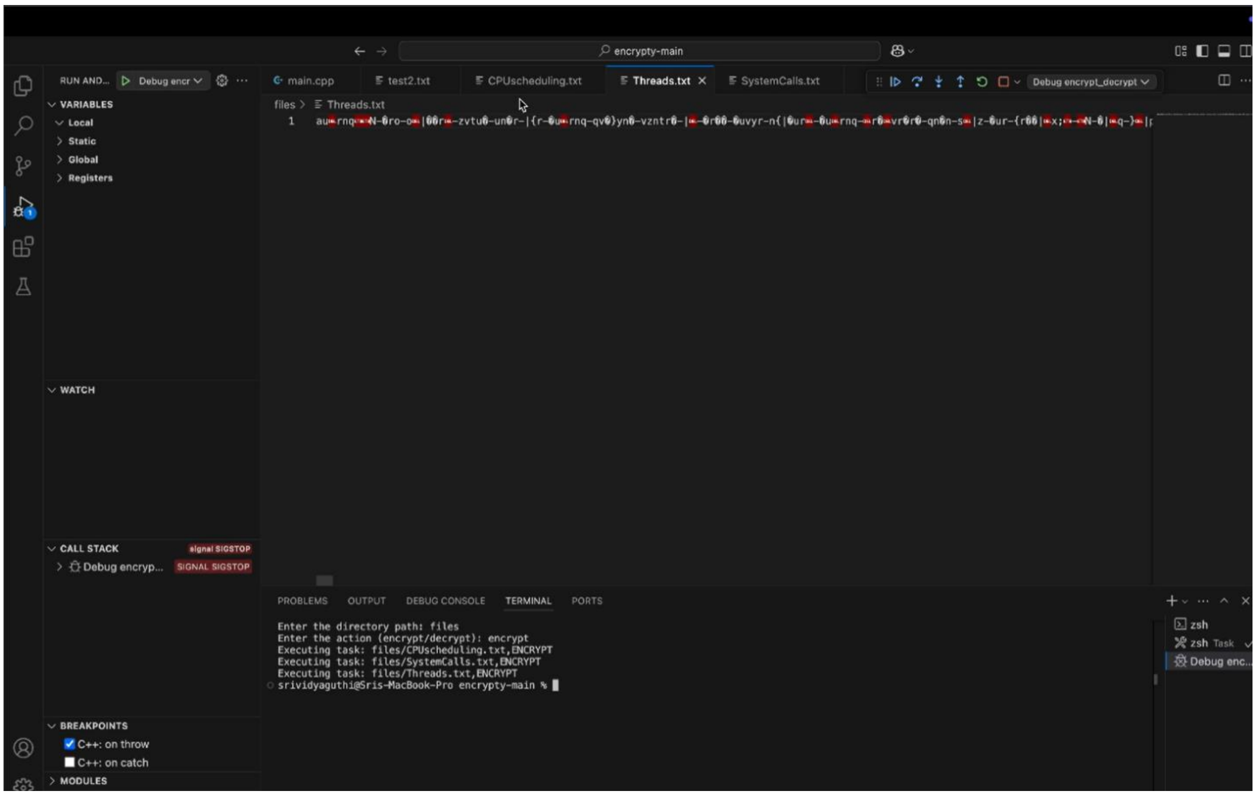
- Input: KhoorZruog
- Decrypted Output: HelloWorld

Screenshots of console output and final file content are included to demonstrate correctness and performance. Execution time was also measured, showing notable speedup using parallelism.



```
CPU SCHEDULING
CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the c
Basic Concepts
In a single-processor system, only one process can run at a time. Others must wait until the CPU is free and can be rescheduled. The object
In multiprogramming several processes are kept in memory at one time. The objective of multiprogramming is to have some process running at
CPU –I/O Burst Cycle
CPU burst: The time a process uses the CPU until it starts waiting for input or is interrupted by another process.
I/O burst: The time a process waits for input/output
The success of CPU scheduling depends on an observed property of processes: process execution consists of a cycle of CPU execution and I/O
CPU Schedule
Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. Conceptually, howev
The selection process is carried out by the short-term scheduler, or CPU scheduler. The scheduler selects a process from the processes in m
CPU-Scheduling decisions may take place under the following four circumstances:
1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wa
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)
3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)
4. When a process terminates.
For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for ex
When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is nonpreemptive or cooperative. Otherwise,
Dispatcher
Another component involved in the CPU-scheduling function is the dispatcher.
The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler.
This function involves the following:
• Switching context
• Switching to user mode
• Jumping to the proper location in the user program to restart that program.
The dispatcher should be as fast as possible, since it is invoked during every process switch. The time it takes for the dispatcher to stop
Scheduling Criteria
Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm is based on the following criteria.
• CPU Utilization. We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. For a lightl
• Throughput. Number of processes that are completed per unit time is called throughput. For long processes, this rate may be one process p
```

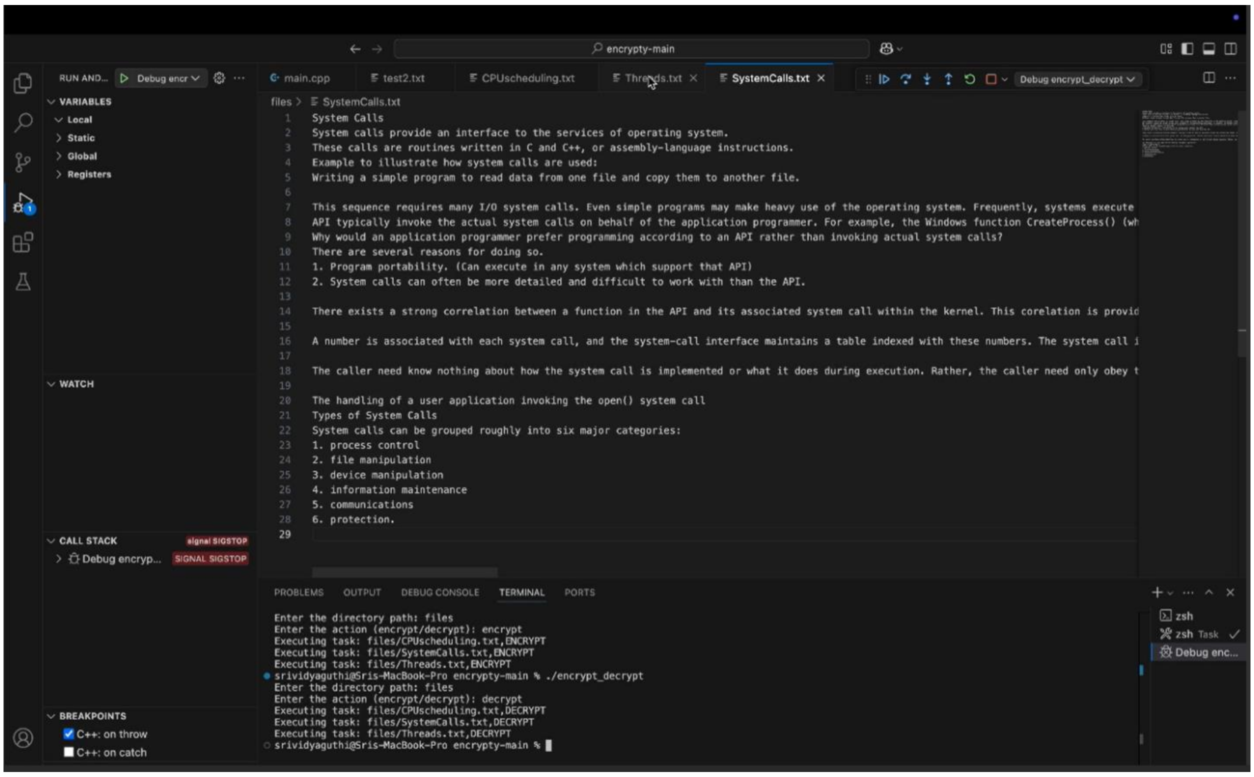
```
srividyaguthi@Sris-MacBook-Pro encrypty-main % 
```



files > Threads.txt

```
au�rnq��N-�ro-o�|��r�-zvtu�-un�r-|{r-�u�rnq-qv�}yn�-vzntr�-|�-�r��-�uvyr-n{|�ur�-�u�rnq-�r�vr�r�-qn�n-s�|z-�ur-{r��|�x;��-�N-�|�q-}�|
```

```
Enter the directory path: files
Enter the action (encrypt/decrypt): encrypt
Executing task: files/CPUscheduling.txt,ENCRYPT
Executing task: files/SystemCalls.txt,ENCRYPT
Executing task: files/Threads.txt,ENCRYPT
srividyaguthi@Sris-MacBook-Pro encrypty-main %
```



files > SystemCalls.txt

```
System Calls
System calls provide an interface to the services of operating system.
These calls are routines written in C and C++, or assembly-language instructions.
Example to illustrate how system calls are used:
Writing a simple program to read data from one file and copy them to another file.

This sequence requires many I/O system calls. Even simple programs may make heavy use of the operating system. Frequently, systems execute
API typically invoke the actual system calls on behalf of the application programmer. For example, the Windows function CreateProcess() (wh
Why would an application programmer prefer programming according to an API rather than invoking actual system calls?
There are several reasons for doing so.
1. Program portability. (Can execute in any system which support that API)
2. System calls can often be more detailed and difficult to work with than the API.

There exists a strong correlation between a function in the API and its associated system call within the kernel. This corelation is provid

A number is associated with each system call, and the system-call interface maintains a table indexed with these numbers. The system call i

The caller need know nothing about how the system call is implemented or what it does during execution. Rather, the caller need only obey t

The handling of a user application invoking the open() system call
Types of System Calls
System calls can be grouped roughly into six major categories:
1. process control
2. file manipulation
3. device manipulation
4. information maintenance
5. communications
6. protection.

```

```
Enter the directory path: files
Enter the action (encrypt/decrypt): encrypt
Executing task: files/CPUscheduling.txt,ENCRYPT
Executing task: files/SystemCalls.txt,ENCRYPT
Executing task: files/Threads.txt,ENCRYPT
srividyaguthi@Sris-MacBook-Pro encrypty-main % ./encrypt_decrypt
Enter the directory path: files
Enter the action (encrypt/decrypt): decrypt
Executing task: files/CPUscheduling.txt,DECRYPT
Executing task: files/SystemCalls.txt,DECRYPT
Executing task: files/Threads.txt,DECRYPT
srividyaguthi@Sris-MacBook-Pro encrypty-main %
```

CONCLUSION

This project effectively demonstrates how parallel processing techniques, specifically **multiprocessing** and **multithreading**, can be applied to enhance the performance of cryptographic operations like file encryption and decryption.

By dividing large files into smaller chunks and processing them concurrently, the project minimizes the overall execution time, making it highly efficient for handling large datasets. Two distinct approaches were implemented:

1. **Multiprocessing** using the fork() system call creates separate child processes for each task. This provides **better isolation** between tasks, as each process has its own memory space. However, this approach incurs higher overhead due to context switching and process management by the operating system.

2. **Multithreading** using POSIX threads (pthreads) allows tasks to share the same memory space. By utilizing **shared memory** and **semaphores** for synchronization, threads can communicate and coordinate efficiently. This results in **lower overhead** and faster execution, especially on systems with multiple cores.

Key Takeaways:

1. **Parallelism Boosts Performance**
   The use of parallel processing significantly improves the throughput of I/O-intensive operations like file encryption and decryption. Processing multiple chunks simultaneously reduces wait time and speeds up completion.

2. **Effective Synchronization is Crucial**
   In the multithreaded model, where threads operate in a shared memory space, **synchronization mechanisms like semaphores** are critical. They ensure that no two threads access or modify shared data simultaneously, thus preventing race conditions and data corruption.

3. **Choosing Between Threads and Processes**
   The choice between multiprocessing and multithreading depends on the specific use case:
   - Use **multiprocessing** when **task isolation** is a priority and shared memory is not required.
   - Use **multithreading** when **resource sharing** and **lightweight execution** are more important than isolation.

4. **Scalability**
   This architecture is scalable and adaptable to various encryption algorithms or even other data-processing workloads. The core concept—divide the task, process in parallel, and combine results—can be extended beyond cryptography.

In conclusion, this project highlights the practical advantages of parallelism in real-world applications and serves as a foundational model for building efficient and scalable encryption systems.

FUTURISTIC SCOPE

The current system effectively uses parallel processing for file encryption and decryption, but there is significant potential for enhancement. Below are five key areas where the project can evolve further:

1. Integration of Stronger Encryption Algorithms

The system can be upgraded to use industry-standard algorithms like AES, RSA, or Blowfish. These provide higher levels of security and make the tool suitable for real-world applications where data protection is critical.

2. Dynamic Load Balancing

Instead of dividing tasks statically, dynamic load balancing can be implemented to distribute encryption/decryption tasks based on available system resources. This ensures better CPU utilization and improved efficiency on multi-core systems.

3. GPU Acceleration

Introducing GPU support (e.g., using CUDA) can allow the system to perform massively parallel operations, significantly reducing encryption and decryption times for large files or high-volume data processing.

4. Cross-Platform and GUI Support

Extending the project to support Windows, along with developing a user-friendly GUI, will make the tool accessible to a wider audience. A graphical interface can simplify usage for non-technical users and improve overall usability.

5. Cloud Integration and Real-Time Encryption

The system can be adapted to support encryption before uploading to cloud storage platforms like Google Drive or Dropbox. Additionally, real-time encryption/decryption for streaming or live file access can be introduced for faster and more secure data handling.

References

1. Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley.

2. Stallings, W. (2018). *Cryptography and Network Security: Principles and Practice* (7th ed.). Pearson.

3. Tanenbaum, A. S., & Bos, H. (2015). *Modern Operating Systems* (4th ed.). Pearson.

4. Kerrisk, M. (2010). *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press.

5. Butenhof, D. R. (1997). *Programming with POSIX Threads*. Addison-Wesley.

6. OpenSSL Project. (n.d.). *OpenSSL: Cryptography and SSL/TLS Toolkit*. Retrieved from https://www.openssl.org/

7. GeeksforGeeks. (n.d.). *Multithreading in C++*. Retrieved from https://www.geeksforgeeks.org/multithreading-in-cpp/

8. TutorialsPoint. (n.d.). *C++ Multithreading and Process Management*. Retrieved from https://www.tutorialspoint.com/cplusplus/index.htm