FATİH SULTAN MEHMET VAKIF ÜNİVERSİTESİ

**Student:**

Name: Enes

Surname: Karaçay

ID Number: 2121221047

Department: Computer Engineering

**Project:**

Topic: Distributed Calculator Project

**Course:**

Name: Operating System

Instructor: Samet Kaya

## Content

## 1- Project Topic : Distributed Calculator System

The Distributed Calculator Project involves the design and implementation of a process-based calculator system that performs four fundamental mathematical operations: addition, subtraction, multiplication, and division. Each of these operations is delegated to a separate subprocess to ensure modularity and concurrency. The main program, referred to as the Calculator, acts as the central controller and user interface.

The project leverages **inter-process communication (IPC)** through unidirectional pipes for data transfer between the Calculator and its subprocesses. The system is designed to demonstrate several core concepts in operating systems, including **process creation**, **synchronization**, and **communication mechanisms**. Here's a detailed breakdown:

1. **User Interaction**: The Calculator provides a menu-driven interface where users can choose a mathematical operation, input two operands, and view the result. The system operates iteratively, allowing users to perform multiple calculations or exit as desired.
2. **Process Management**: The Calculator program employs the fork system call to spawn child processes. Each subprocess is dedicated to a specific operation and is executed using the exec family of system calls. This approach separates concerns and allows independent execution of operations.
3. **Communication via Pipes**: The Calculator communicates with each operation subprocess using dedicated pipes. Operands are sent from the main program to the subprocess, which performs the calculation and sends the result back.
4. **Result Persistence**: A dedicated program, Saver, is invoked by each operation subprocess to persist calculation results in a file (results.txt). Results are passed to the Saver program via command-line arguments, ensuring efficient and streamlined storage without additional pipe overhead.
5. **Robust Error Handling**: The system includes mechanisms to handle common errors, such as invalid inputs, division by zero, and communication issues, to provide a smooth and user-friendly experience.
6. **Technical Foundation**: The project is implemented in the **C programming language** for a Linux-based environment. It demonstrates proficiency in essential Linux and C programming concepts, such as:
    - Process creation and termination (fork, exec, wait).
    - Inter-process communication via pipes (pipe, read, write).
    - Dynamic memory management and robust error handling.

The Distributed Calculator Project is a practical demonstration of how operating system concepts can be applied to develop efficient and modular software systems. Its design and implementation emphasize concurrency, modularity, and system reliability, providing a hands-on understanding of advanced programming concepts.

## 2- Tasks Completed During the Project

The Distributed Calculator Project required the implementation of several interdependent components to achieve a fully functional distributed system. Each task was carefully planned and executed to meet the project requirements. Below is a detailed explanation of the tasks completed:

### **1.** Main Program (**Calculator**) Development

- **Menu Creation**: The main program was designed to display a menu-driven interface, allowing the user to select a mathematical operation (addition, subtraction, multiplication, division) or exit the program.
- **Input Validation**: The program includes error-handling mechanisms to validate user inputs. For example, checks are performed to ensure that users select valid operations and provide valid numeric operands. Additional checks handle division by zero.
- **Process Creation and Management**: Using the fork system call, the program spawns child processes for each mathematical operation. The exec family of system calls (execve) is used to replace the child process's image with the respective operation's code (e.g., addition, subtraction).

### **2.** Subprocess Implementation

- **Mathematical Operation Subprocesses**: Four separate C programs were implemented—addition.c, subtraction.c, multiplication.c, and division.c. Each program:
    - Reads operands from its respective pipe.
    - Performs the specified mathematical operation.
    - Sends the result back to the main program through a result pipe.
- **Error Handling in Subprocesses**: Subprocesses handle edge cases, such as incorrect data received or division by zero.

### **3.** Inter-Process Communication (IPC) Setup

- **Pipe Creation**: Five pipes were set up to facilitate data transfer between the Calculator and its subprocesses: one for each mathematical operation and one for returning results.
- **Data Transfer**:
    - The main program sends operands to the appropriate subprocess via pipes.
    - Subprocesses compute the result and return it to the Calculator through the result pipe.
- **Synchronization**: Proper synchronization was ensured using the wait system call so that the main program waits for the subprocesses to finish execution where needed.

### **4.** Result Persistence with the **Saver** Program

- **Result Passing**: Each operation subprocess invokes the Saver program upon completing a calculation.

- **File Writing**: The Saver program receives the calculation result as a command-line argument and appends it to a file (results.txt).
- **Data Integrity**: The Saver program ensures that results are correctly written to the file without overwriting previous entries.

## **5.** Makefile Creation

- **Automatic Compilation**: A Makefile was created to streamline the compilation of all components (calculator.c, addition.c, subtraction.c, multiplication.c, division.c, and saver.c). Running make compiles the entire project, producing executable files for each program.
- **Ease of Use**: The Makefile ensures that all dependencies are compiled correctly and in the right order, reducing manual effort and potential errors.

## **6.** Testing and Debugging

- **Functional Testing**: The system was tested for correctness by performing various calculations and comparing the results with expected outputs. All edge cases, such as division by zero, invalid inputs, and simultaneous operations, were evaluated.
- **Debugging**: Issues like pipe communication errors, data corruption during transfer, and subprocess synchronization were identified and resolved.

## **7.** Documentation and Reporting

- **User Guide**: A detailed report was created to document the project, including explanations of the workflow, implementation details, and examples of program usage.
- **Screenshots**: Screenshots of the program in action were taken to demonstrate the functionality of the Calculator.

## 3- Additional Notes

### **1.** Input Validation

- **Operation Selection**: The main program rigorously validates the user's input to ensure that only valid choices (1 to 5) are accepted. This prevents unexpected errors by guiding the user to select a correct option from the menu. Invalid inputs trigger informative error messages and prompt the user to try again without causing a program crash or abnormal termination.
- **Operand Validation**: For mathematical operations, the program ensures that only valid numeric inputs are accepted as operands. This prevents computational errors and ensures the integrity of the data processed by the system. If invalid input is detected, the program displays a relevant message and waits for correct input.

### **2.** Division-by-Zero Handling

- **Safety Check in Division Subprocess**: The division subprocess includes checks to ensure that the second operand is not zero, thus preventing an illegal division operation. If the user attempts to divide by zero, the system immediately notifies them of the error and does not attempt the calculation. This safeguard maintains program stability and prevents undefined behavior.

### **3.** Resource Management

- **Pipe Management**: All pipes are carefully managed to ensure they are closed appropriately in both parent and child processes once they are no longer needed. This helps prevent resource leaks that could degrade system performance. Proper pipe closure is essential for maintaining good inter-process communication and system resource availability.
- **Process Termination**: The program ensures that processes terminate cleanly when the system exits. This prevents orphan or zombie processes from lingering, which could consume system resources unnecessarily. Using proper termination signals and the wait system call ensures that all child processes complete execution before the main program exits.

### **4.** File Management

- **Result Persistence with Saver**: The Saver program ensures that all results are stored persistently in the results.txt file. Each result is appended rather than overwritten, allowing users to view a comprehensive history of calculations. This ensures data consistency and long-term tracking of computation outcomes.
- **File Access and Integrity**: The system uses standard file operations to open, write, and close the file securely. Proper error handling is implemented to alert the user in case of any file access issues, preventing data corruption or loss.

## **5.** Scalability

- **Extensible Design**: While the system currently handles four basic mathematical operations, its architecture supports easy expansion. New operations can be added by creating additional subprocesses, defining new pipes, and integrating them into the main program. This modular design promotes scalability and future enhancement without major redesign efforts.

## **6.** Error Handling

- **General Error Messages**: The system includes comprehensive error-handling routines that provide clear feedback for issues such as:
    - **Invalid Input**: The program rejects invalid inputs and prompts the user with a clear message, ensuring the program remains in a stable state.
    - **Division by Zero**: The division operation checks for zero as the second operand and responds with an error message to inform the user that this operation is not allowed.
    - **Pipe Communication Failures**: Errors related to pipe creation or communication are captured and logged, ensuring that the user is aware of any communication issues. This provides robustness by identifying points of failure during inter-process communication.

## **7.** User Experience and Usability

- **User Feedback**: The program is designed to provide real-time feedback for each input and operation. If an error occurs, users are informed immediately and guided toward corrective action.
- **Menu Navigation**: A simple, intuitive menu allows users to easily choose operations and input values, making the program suitable for users with varying technical experience.

## **8.** Code Quality and Documentation

- **Commenting and Code Clarity**: Each source code file is thoroughly documented with comments explaining the purpose of key sections and functions. This helps anyone reviewing or maintaining the code to understand the logic and structure of the system.
- **Consistent Style**: The code adheres to a consistent style, improving readability and maintainability. A well-structured codebase makes future modifications and debugging more efficient.

## 4- References

- Linux Pipe and Fork Usage: https://man7.org/linux/man-pages/man2/pipe.2.html
- execv Fonksiyonu Hakkında: https://man7.org/linux/man-pages/man3/exec.3.html
- theory and laboratory lecture notes.
  - I found a github project similar to this project:
- https://github.com/Adam-Kosicki/Calculator-using-Pipes-and-Child-Processes