Bank Mortgage Calculator System.

September 2022

Kareen Ziadat

Contents

MAIN SYSTEM DESCRIPTION 3
THEORETICAL BACKGROUND 4
CODING 12
SHELL SCRIPTING VS C LANGUAGE 15
THREADING 20
FINALIZING 25
REFERENCES 32

Main system description

My project aims to provide a bank with a mortgage calculator to provide clients with their monthly payment and provide the bank with statistics on the monthly payments. The bank worker inputs the principal amount, yearly interest rate, and number of payments and the server will calculate and give the monthly payment (this will be given to the person getting the mortgage by the worker). The system allows a customizable number of clients that will be asked once the connection is established that the worker can specify. The server will provide its service to multiple clients concurrently due to threading.

It consists of 3 components:

1. Server: this component provides the mortgage calculator and writes the client information and their monthly due payment to a file. The server is always on and provides its service to the clients concurrently. After all clients have gotten their monthly rate, the server will provide the bank worker with an analysis of minimum payment, maximum payment, and the average payment on the terminal.
2. Client: The worker will enter the first and last name of the client, the principal, interest rate, and the number of payments and will get the amount that client should pay every month.
3. Statistics: this code will provide the bank with statistics on minimum, maximum, and average monthly payment of clients that can be used later.

The server and client communicate over the network using sockets following the TCP/IP protocol. The server writes the information about the clients to a file that is read from in the statistics file. The statistics code is run using the exec command that is called once no more clients are connecting to the server. The server, however, is always running.

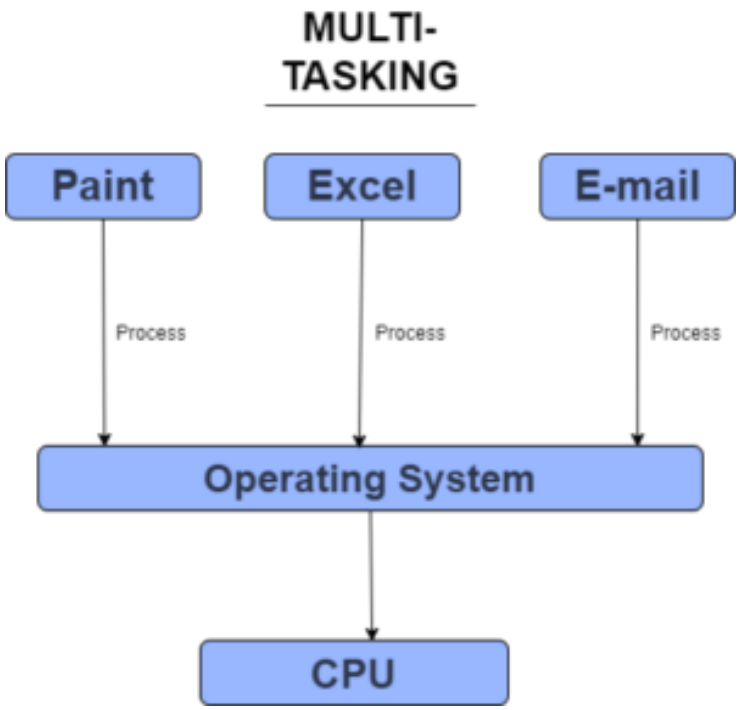
Theoretical background

What is a Process?

A process is a key concept in an operating system that describes program execution on a computer allowing multitasking and resource management. The execution of a program starts a process, either by input via a shell command or by the execution of another program. To complete tasks efficiently, code, data, and system resources are handled within the process.

A process is an isolated and independent sequence of execution that the kernel of an operating system treats as a single entity for the purpose of utilizing system resources (memory space, I/O devices, CPU time). A data structure known as the Process Control Block (PCB) uniquely identifies each process, allowing the operating system to maintain oversight and control over each process's status and resource allocation. Even when numerous processes run concurrently on the same system, exclusive access of processes is maintained, preventing unintentional interference between processes, and contributing to system stability and security.

Multiple processes enable concurrent task execution, process isolation (improved stability and security), effective system resource management, improved fault tolerance (through failure containment), scalable system architecture, parallel processing (enhanced performance), streamlined load balancing, and the promotion of interoperability among diverse software components, all of which contribute to overall system efficiency and reliability.



How OS-controlled resources are used among processes of different purposes.

Different mechanisms are used in an operating system to ensure fair and efficient resource use, such as CPU time, memory, disk space, and I/O devices (drives or network interfaces).

1- Memory management

Where a computer stores data and instructions that are in use. It determines how much data a system can keep while in use and having enough memory enables better multitasking, faster program loading, and overall better system performance.

The operating system allocates memory space for the new process, which includes the program's instructions (Code), data, and stack. It controls memory allocation by dividing it into fixed-size chunks (pages) or assigning variable-size blocks. It also guarantees that processes can only access memory areas that are assigned to them. When a process ends (completes execution or faces an error), the OS releases the process's resources, updates appropriate process control blocks, and may notify other processes of its termination. Through system calls, users can dynamically allocate and deallocate memory using malloc, realloc, and calloc.

2- CPU time

The amount of processing time dedicated to running a program or performing a certain task. It has a direct impact on the speed and responsiveness of a computer system and efficient CPU time allocation ensures that tasks are finished on time, avoiding slowdowns.

The CPU cannot execute all processes simultaneously, so the OS uses CPU scheduling algorithms to choose which task should run at any given time. The OS switches between processes quickly to provide the illusion of parallelism enhancing effectiveness.

3- I/O devices

Processes frequently interact with I/O devices (keyboards, mouse, displays, printers, hard drivers, SSD, etc.). They allow interaction with the computer as well as data storage and retrieval. Efficient I/O operations are critical for user experience and data management as slow I/O can cause delays in data reading/writing, reducing overall system responsiveness.

The operating system manages access to these devices and provides abstractions such as device drivers and APIs. Moreover, it maintains device queues and guarantees that processes have equal access to devices.

4- Disk space management

I/O operations like reading from or writing to disk and interacting with devices are frequently required by processes. The OS controls these tasks by allowing processes to interact with devices via APIs, buffering data to registers to improve I/O efficiency, and providing file management functions that allows actions like opening, reading, writing, and closing files. The OS also ensures that several processes can access files concurrently without conflict.

5- Network Bandwidth

When processes need to communicate with one another via the network, the OS handles network resources like bandwidth allocation. It ensures that processes can send and receive data without congesting the network using network protocols like TCP/IP, UDP, and other algorithms.

Synchronization and communication
To guarantee smooth operation inside a complex system, processes must frequently coordinate or communicate. This is needed to allow information, resources, and interdependent actions to exchange. This coordination improves efficiency, quality, and fulfills tasks.
The OS provides synchronization methods for inter-process communication (IPC), like mutexes, and message forwarding, to ensure that processes can coordinate their operations and share data without conflict.

Overall, the operating system serves as a resource manager, coordinating and arbitrating access to system resources among different purposes processes.

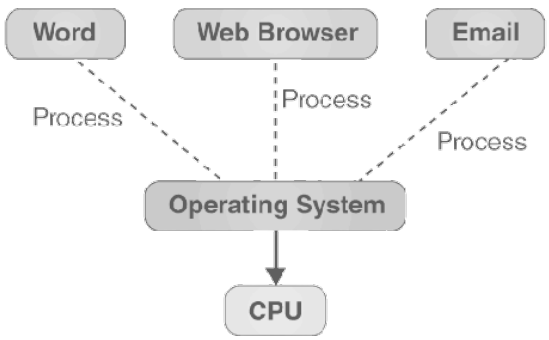
Multitasking and concurrency

Multitasking and concurrency are fundamental concepts in operating systems. They allow management and execution of several tasks or processes at the same time. Multitasking refers to the operating system's capacity to juggle many tasks and make them appear to be running concurrently, even on a single CPU, by swiftly switching between them. Concurrency, on the other hand, is concerned with the broader concept of executing several activities concurrently, which includes scenarios in which processes might run in parallel, share resources, and require synchronization. Multitasking and concurrency are critical for enhancing system performance, responsiveness, and resource utilization.

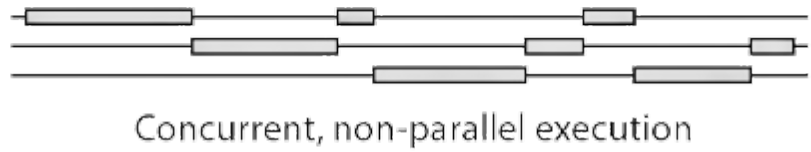
Multitasking (multiprocessing)

Simultaneous execution of numerous processes or tasks. The phrases multitasking and multiprocessing are frequently used interchangeably, however multiprocessing implies the involvement of more than one CPU. On the other hand, when only one CPU is involved, it switches from one process to another so quickly that it appears to be running all the programs at the same time.

A frequent use case for multitasking is running multiple applications on a personal computer.



Concurrency



Concurrent, non-parallel execution

Concurrent, parallel execution

Concurrency refers to the idea of a single process handling many tasks at the same time by breaking the process down into separate threads (a lightweight process that can be executed separately from other threads) that can run independently. Concurrency can be implemented on to a single or multiple CPUs. Concurrency is frequently used to boost the speed of a single application by allowing several components of the application to execute concurrently. Several issues need to be addressed such as race conditions, deadlocks, and hunger, and must be handled (not always).

A frequent use case for concurrency is a web server that handles several client requests concurrently.

Key differences

	Multitasking	Concurrency
CPU Utilization	Single CPU with time-sharing or multi-core CPUs	Can involve single or multiple CPUs
Parallel Execution	Can involve true parallel execution (in multi-core CPUs)	May or may not involve true parallelism
Task Interaction	Tasks may or may not interact directly	Tasks often interact and may require synchronization
Resource Management	OS allocates CPU time to tasks	Synchronization methods are frequently required to manage shared resources among threads.
Performance Consideration	May have context switching overhead when switching between tasks.	May have contention and synchronization overhead because of thread interactions.
Task Independence	Tasks independent to each other	Tasks can be interrelated or dependent
Resources	No shared resources	Shared resources

Both concepts seek to maximize computing resources while allowing numerous jobs or processes to execute concurrently.

In my code:

Code1: multitasking using separate processes.
Code 1 divides the application into distinct processes by using the fork system function. The primary process (parent process) prompts the user to enter ten array elements. After that, each child process is created to perform its assigned function (calculate average, find minimum value, and sorting the array). These child processes send their results back to the parent process via pipes, with one pipe sending the average value, another the minimum value, and another the sorted array. The use of pipes is needed to facilitate inter-process communication between the processes. Pipes enable efficient data sharing because each process operates within its own memory region (independent). This method ensures that the CPU can move between processes quickly, allowing for effective multitasking capabilities.

Code 2: demonstrates concurrency using threads.
Code 2 employs a multi-threaded approach, where individual threads carry out different tasks (calculating minimum, average, and sorting) within a single process. Since threads work concurrently and share the same memory space, CPU utilization is high. Moreover, since threads share this memory space, inter-process communication is not required, hence pipes are unnecessary. Furthermore, because there are no competing data, this method does not require synchronization.

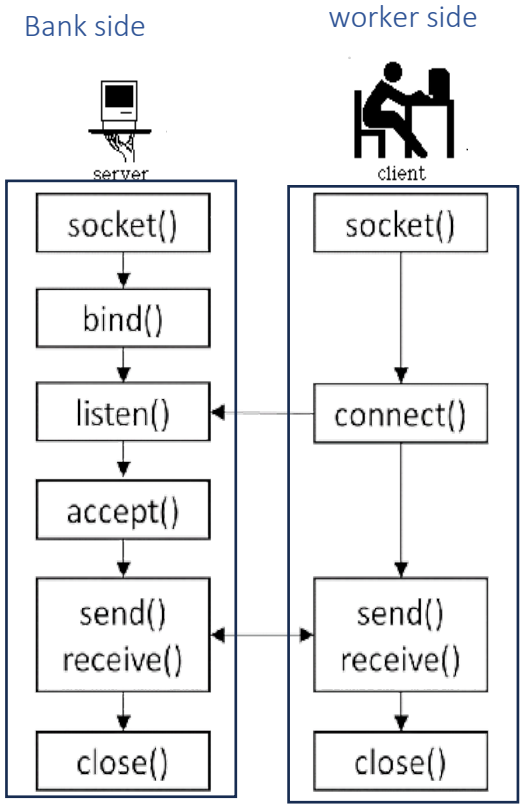
Both code 1 and 2 achieve parallel execution but in different ways where code 1 creates separate processes and code 2 by creating threads. Multitasking has a higher overhead for creating and managing multiple processes compared to creating and managing threads.

Difference summarized:

	Code 1 (Multitasking)	**Code 2 (Concurrency)**
CPU Utilization	Low	High
Parallel Execution	Yes	Yes
Task Interaction	Inter-process	Intra-process (threads can share data directly)
Resource Management	Multiple processes	Threads within a process
Performance Consideration	Overhead of process creation	Lower overhead for thread creation
Task Independence	Independent processes	Threads share memory and resources simplifying data sharing.

Communication ability of different processes

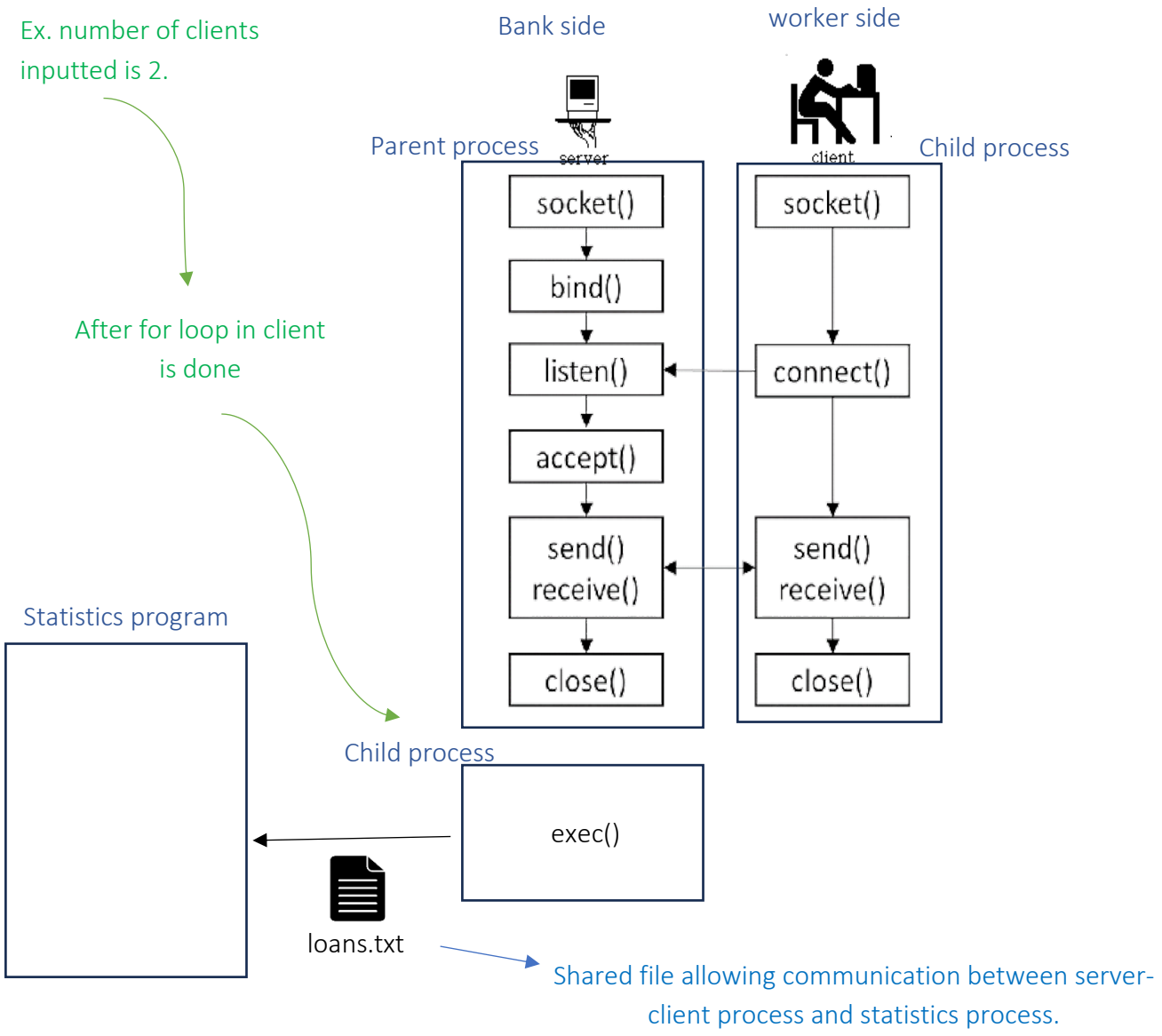
Since the bank's server is located remotely, the bank workers need to connect to it through sockets. Therefore network programming plays a crucial role in facilitating the communication between these processes and is established using TCP (Transmission Control Protocol).
A TCP connection is formed when a worker connects to the bank server via the network to access the mortgage calculator service. For testing purposes, I am using the IP address 127.0.0.1 (localhost) and specifying the port number as the same for worker and server to allow communication. TCP ensures that data is delivered reliably, in the correct order and will retransmit any packets that are lost during transmission, ensuring that the data reaches fully. TCP is a connection-oriented protocol; it establishes, maintains, and terminates a connection between the client application and the bank server.
The socket programming interface is required for network communication to enable processes on separate machines to connect and exchange data. In my case, the client program would open a socket and link it to the bank server's socket on the localhost and port, allowing bidirectional communication.
The client-side socket requests the connection, whereas the server-side socket listens for incoming connections on the port specified. Both sides can transmit and receive data after the link is established.
For simulation purposes, I have forked my code, let the server run in the parent and the client to run in the child process (they are independent) therefore socket is needed. Ssystem calls used to establish socket is shown



Parent process in my case Child process in my case

fork()

After every client, the server saves their full name and month payment onto a file which is later sent to the statistics program.
Once all the clients are done (the number of clients asked for at the begging of the server), the CPU starts running the other child process which includes an 'exec' function.
This child process is responsible for running the statistical program. To do so, it uses an exec function to execute the statistic program and sends the file created by the parent (containing full names, monthly payment) as an argument.
In this part, I've used the fork system call to create two independent processes, as it is recommended when using the exec function system call. The exec system call family is used to replace the program run by a process. When a process runs exec, it loses all its current code and data and is replaced with the executable of the new application. I want to make sure the parent process is always running since it has the server code (should always be up).

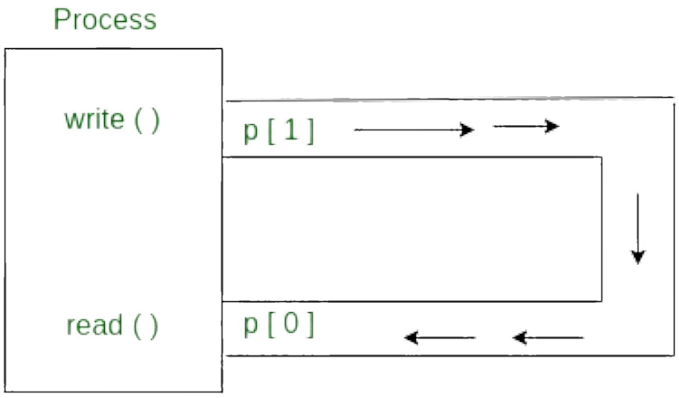


Let us take a closer look at the statistical program. To improve its efficiency, I used the fork function to create multiple child processes, each of which is responsible for generating different statistics (such as minimum, maximum, and average). I've set up one-way pipes to allow communication between these child processes and the parent (since they are independent).

When creating the pipes, two descriptors are returned - one for reading and one for writing. The reading descriptor allows reading data from the pipe, while the writing descriptor allows writing data into the pipe.

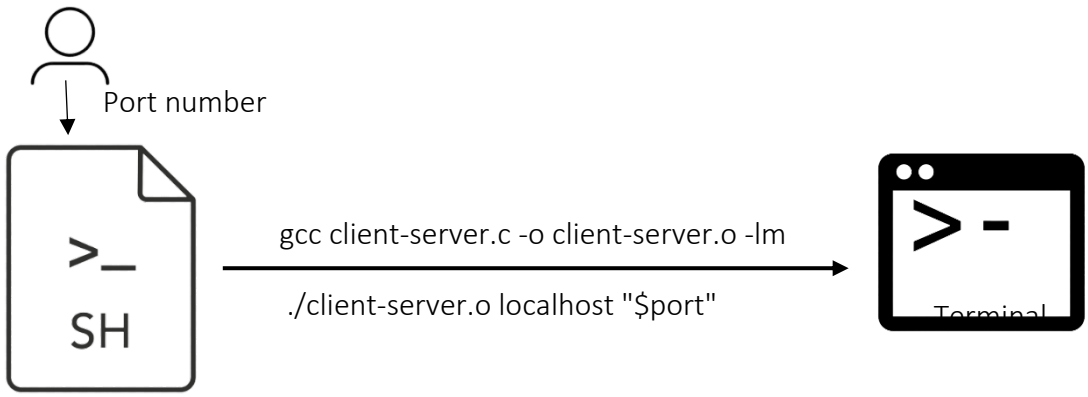
Each child process is given the writing end of the pipe, while the parent's process is given the reading end. This allows the child processes to write their computed statistical results into their end of the pipe. The parent can then read this data by reading from its end of the pipe. So, each child process computes its statistic, then writes the result to its file descriptor for its pipe. This result is transferred through the pipe to the reading end that the parent has access to. The parent can then read the results from each child by reading from the file descriptors it has access to. This allows seamless one-way communication from each child to the parent.

The parent reads the results from each pipe and prints them to the terminal, completing the transfer of data from child processes to parent process using pipes and file descriptors for reading and writing.



This image only depicts one child writing to the parent, there will be four pipes (min, max, avg, count).

I have also employed shell script. My shell script allows three options, running the mortgage system (inter-process communication), searching for client information, and changing the permissions of the text file that has the loan details. The first option will ask the user for the port number, compile and run the system (since the server is in the parent the CPU will start executing it first ensuring that the client can connect to it.) the second option uses the GREP function to search for a specific client and show their monthly payment. Lastly, it allows changing the permission by using the chmod.



Coding

Code name (part1task4).

System requirement: allow users to sign agreement to terms and conditions of university and ensuring they follow a specific format provided by the university.

This program is designed as a multi-process application that leverages various system calls and file operations to handle the system requirement. Its primary goal is to process a text file while also facilitating communication across processes. The program uses system calls like fork, pipe, wait, and execlp, as well as file descriptors for file management and manipulation.

The program begins with the creation of a pipe, which serves as a communication channel between parent and child processes. Following that, a child process is created using the fork system function. The child is responsible for reading an input text file and replacing the placeholders with user-supplied information.

The CPU starts by running the parent code which outputs a message (You will be asked to enter your name and the date to sign the terms and conditions form.) Then it reaches the wait(NULL) system call so it executes child and waits till it finishes execution to continue the parent process.

The code relies heavily on file descriptors to accomplish its goal. To begin, it uses the O_RDONLY flag to open the input file and retrieve a file descriptor (fd_input) in the child. Simultaneously, it generates a file descriptor (fd_output) by creating a new output file (or truncating an existing one) with the O_WRONLY, O_CREAT, and O_TRUNC flags in the parent.

Within the child process, the program opens the input file, prompts the user to enter their name and the date, and then tokenizes and processes the file's content. If the program finds the placeholders ([name] or [date]) it writes the user-supplied information (their name or data) to the pipe; otherwise, the word itself is passed to the pipe. The child process concludes by closing file descriptors (pipe and file) and exiting.

Returning to the parent process, it opens the output file, reads from the pipe (which contains the updated text), and writes it to the output file. It then closes all file descriptors and uses the wait system call to wait for the child process to finish. Following that, it prints a message (You agreed to the terms and conditions, thank you) and waits a brief 3-second pause using sleep. (So user can read this message)

Finally, a new child process is created with fork to execute the clear command, which is responsible for clearing the terminal screen. The execlp system call is used within this child process to replace the current process image with the clear command, ensuring that the terminal screen is cleared. An error message is produced if the "execlp" call fails. The parent process then waits for the clear command to complete before exiting.
Essentially, this program demonstrates the use of file descriptors to read, manipulate, and manage files in conjunction with inter-process communication using pipes (the independent processes are created using the

fork()). The pipe acts as a vital communication channel between the parent and child processes, allowing the child process to modify the file's content and pass it to the parent who will write it onto the output file. Furthermore, synchronization is accomplished by wait() methods, which ensure that processes are executed in the desired order. The program also uses run system commands such as clear using execlp(), demonstrating its versatility in controlling both file operations and external processes.

Note: In another code (client-server model mentioned in the precious task) I've also used an exec function to run the statistics program using exec().

Evaluation of the effectiveness of the design and component

Requirements:

1. Remote Accessibility: Users should be able to access the placement test from different locations.
2. Execution time: minimizing waiting and processing times.
3. Fault Tolerance: If an issue occurs for one user, it does not disrupt the entire system.
4. Scalability: accommodate to many users without significant performance degradation.
5. User-Friendly: The interface for clients should be easy to follow and understand.
6. Data Integrity: Ensure monthly payment results are accurate.
7. Concurrent Access: Multiple clients should be able to access the exam concurrently without conflict.
8. Multiple users: more than one client can use the calculator.
9. Service availability: the server should be available all the time when needed for access.
10. Adaptability: the system should be easily modified to reflect business changes.

In the design and implementation of this server-client mortgage calculator system, several key components and strategies will be employed to meet requirements effectively in terms of efficiency and execution time. First, the server, which hosts the calculator, should be placed on a separate machine from the workers. This separation ensures that workers can access the calculator remotely, enhancing accessibility and convenience. The server will be assigned a known IP address and a specific port number, enabling seamless connection. This setup is crucial because it restricts access to the calculator and the loan file, making it impossible for unauthorized clients to circumvent the system's security measures.

Sockets will be created at the worker's machine and the server machine to allow over the network connection. The worker's machine will initiate communication by setting up a socket for sending and receiving data, while the server machine, which processes and responds to client requests, will also create sockets.
In my case, I plan to use the loopback network interface on the local system for simulation purposes. The loopback network (localhost or 127.0.0.1) is a type of network interface that allows communication to take place within the same machine. In other words, data sent through this interface is not routed through an external network, but rather remains within the local system.

To facilitate concurrent access and optimize efficiency, the server will employ a threading model. Threads allow multiple workers to access the calculator simultaneously, rather than serving them sequentially, thus

reducing waiting times. This multi-threaded approach not only enhances performance but also bolsters fault tolerance. In the event of an issue faced by one worker, the entire server should not go down, preventing a single point of failure. This design choice will decrease execution time and enhance efficiency of the system.

I should also make sure to keep the server running and offering its service all the time. I will also make sure to allow multiple workers to access the server at the same time (every time a different number of clients). This ensures efficiency by ensuring the server is accessible whenever a worker wants to access it.

I will implement this system to decrease the calculation overhead from the worker's machine and let the server carry out this calculation. The server has better capabilities which will decrease execution times, enhancing the efficiency of the entire process.

I will ensure that if the bank decides to calculate the monthly payment using a different equation or modify the process, that this will be easily done and wouldn't affect the client. This makes the system efficient by providing a high degree of flexibility and modularity in the system. It also allows changes to be done quickly and independently, decreasing the chance of disruptions or compatibility concerns.

The system should emphasize user-friendliness by providing an easy user interface with clear instructions at each step, ensuring that users never feel stuck, and they get feedback on their payment at the end.

In conclusion, the design and implementation of this server-client system demonstrates a thorough approach to meeting user needs. The separation of the server and the clients, TCP, the use of threading for concurrency, and the use of forks to create two separate processes (client and server) all contribute to a system that is both resilient, user-friendly, and fitting with the stated requirements for a successful mortgage bank system.

Shell scripting vs C language

Shell scripting is a command interpreter that converts text entered at the command line into actions, allowing for the customization of a Unix session as well as the manipulation of operating system features and processes. The most popular shell is "bash," which is also a scripting language. This makes shell scripting a strong tool for utilizing the shell's capabilities and automating numerous operations that would otherwise require a lengthy sequence of commands. On the other hand, C programming language is a versatile and widely used programming language known for its efficiency and low-level hardware control, making it suited for system programming.

Advantages and disadvantages when it comes to system services (file manipulation, process management, text processing):

1. File Manipulation:

C language:

C allows low-level control, speed due to its compiled nature, and the ability to construct unique file formats and data structures. However, it has drawbacks such as greater complexity and the possibility of error-prone manual memory management, which can lead to issues such as memory leaks and buffer overflows.

Shell scripting:

Bash is easy to use in file manipulation tasks and is readable (due to their concise syntax). Furthermore, shell scripts are generally portable across Unix-like computers. However, they often execute slower affecting efficiency in resource management, and they are not well-suited for low-level file manipulation tasks (limited low-level control capabilities).

2. Process Management:

C language:

C provides fine-grained control over process operations and high efficiency due to low memory and execution time overhead. However, it is complex, error-prone, and platform reliant.

Shell scripting:

Shell provides built-in tools such as ps, and kill, for effective process management, as well as the ability to simplify important process management tasks such as process initiation and termination. However, it may not provide the same granular control as languages like C in complex process management scenarios and has a high performance overhead when handling several processes.

3. Text Processing:

C language:

C allows the creation of custom text processing algorithms. Because of its low-level nature, it is also fast, making it suitable for large-scale text processing. However, it is more complex when compared to scripting languages, and offers limited built-in text manipulation functions.

Shell scripting:

Shell scripting offers built-in text processing tools such as sed, awk, and regular expressions for efficient text manipulation. It is also simple, making it suitable for common tasks. However, it isn't very customizable and has poor performance when dealing with extremely big text files.

Summarizing main differences between programming and scripting languages:

1. Efficiency and Performance:

Programming languages like C are known for their greater power and speed compared to scripting languages. C starts as source code and is compiled into executables but isn't easily transferable between operating systems. Additionally, C outperforms shell scripts in terms of efficiency and performance because C code is compiled directly into machine code, leading to faster system service execution times. In contrast, shell languages involve runtime interpretation, potentially resulting in slightly slower execution.

2. Low-Level System Access:

C language allows direct memory management and assembly-level operations, making it well-suited for hardware and system resource control. In contrast, shell scripting operates at a higher level of abstraction, excelling at interacting with system services through built-in commands and utilities.

3. Portability:

C code requires separate compilation for each platform, reducing its portability to computers with compatible architectures. In contrast, shell scripting is highly portable as it relies on built-in shell commands, which are widely available on Unix-like platforms.

4. Rapid Development:

C code can take longer due to its syntax, memory management, and low-level complexities handling. Shell scripting provides a faster development process due to its simple syntax and built-in commands.

5. Complexity and Control:

C language offers significant control and flexibility, making it suitable for complex resource-intensive tasks. On the other hand, shell scripting is better suited for simple system support jobs due to its ease of use and comprehension but may not be the best choice for complex projects.

6. Error Handling and Debugging:

C includes advanced error handling methods and debugging tools that make it easier to identify and resolve problems, especially when compared to debugging shell scripts, which can be difficult and less robust.

Overall, shell scripting is the best option for quick automation of simple system service, text manipulation, and compatibility, but C is better suited for rigorous hardware management, greater control, and increased performance. It also depends on the task that needs to be accomplished.

To further point out the difference between shell scripting and C language I have written two codes rectangle.sh and rectangle.c that both do the same functionality which is calculating the area of a rectangle from the user's input width and length.

rectangle.sh

```
#!/bin/bash

echo "Enter the width of the rectangle: "
read width

echo "Enter the length of the rectangle: "
read length

area=$((width*length))

echo "The area of the rectangle with width $width and length $length is $area"
```

rectangle.c

```c
#include <stdio.h>

int main() {
    double length, width, area;

    printf("Enter the width of the rectangle: ");
    scanf("%lf", &width);

    printf("Enter the length of the rectangle: ");
    scanf("%lf", &length);

    area = length * width;

    printf("The area of the rectangle is: %.2lf square units\n", area);

    return 0;
}
```

Differences between the code:

	Shell Script (code 1)	C Programming (code 2)
Language	Bash shell scripting	C
Syntax	Human-readable, simple	Complex, structured
Data Types	Loosely typed	Statically typed
Arithmetic	$(( ))	Standard C operators
Input/Output	read, echo	scanf, printf
Mathematical Operations	Simple arithmetic	Standard operators
Compilation vs. Interpretation	Interpreted	Compiled (GCC)
Error Handling	Limited error handling	Robust error handling
Portability	Highly portable	Platform-dependent

Shell scripting

The code is intended to implement a system component with a variety of features. To begin with, it contains a function named run_server that asks the user to input the port number. It then compiles the client-server.c program into an executable named client-server.o, with the -lm option informing the GCC to link the math library (libm). After, it runs the compile program client-server.o using the port number parameter specified by the user and the localhost parameter which connects to the same machine using a loopback network. This part of the code will simplify the starting of the client and server program.

Second, there is a search_file method that allows the bank to search for a client within the loans.txt file to get their monthly payment. It prompts the worker to enter the client's name and the script uses the grep tool to run a case-insensitive search and then shows the results.

The script also contains a change_permissions method. This function asks the user for a numerical authorization format and verifies the input to ensure that it conforms to the format of three digits ranging from 0 to 7. If the input format is correct, the script will use the chmod command to change the permissions of the loans.txt file and will offer feedback on the successful permission change.

The script allows the administrator to select any choice from a menu, each of which corresponds to a specific purpose. The user inputs their choice and based on the selection, the script executes the relevant function.

This shell script encapsulates the functionality of the mentioned system component, providing users with a menu-driven interface to adjust file permissions, perform advanced regular expression searches, and handle inter-process communication using shell scripting. File permissions are controlled within the change_permissions function, where the loans.txt file's permission can be changed using the chmod command. Advanced regular expression searches are performed in the search_file function using the grep command. Inter-process communication is managed when the run_server function is invoked from the main menu, as it runs and compiles the (client-server), illustrating inter-process communication between the parent (server) and the child (client).

Threading

The goal of the bank system's server component is to allow several clients to access the calculator at the same time while decreasing waiting times and improving fault tolerance.

First, I will include the pthread.h library which will provide the essential functions and data types for working with threads. Following that, I will write a thread helper function with a void return type and void arguments, encompassing the server's operations, such as handling client data, performing calculations, displaying payment details, and writing information to a file. I will ensure appropriate casting is done to the argument of this thread function (socket file descriptor). I will then declare the thread variable pthread_t in the parent process. I will also use the thread_Create function to create the threads for each client with the appropriate arguments after the client's connection is accepted. The arguments will be the thread variable, NULL, the helper function name and the socket cast to void (in that order).

After all these steps, every time a client is accepted to connect to the server, a new thread is created (using the same thread variable) and the CPU will be able to monitor all these threads extremely quickly that is appears that ever client is accessing the server independently (server code is repeated for each thread/client). The goal is to have more than one session to facilitate concurrent service for multiple clients, enhancing performance. Without threading, each client that connects will reserve the entire socket causing sequential execution and in case this client enters an infinite loop or encounters and error, it stops the entire system from functioning creating a single point of failure (any issue on this client will affect all customers waiting sequentially).

Discussing Results of Thread Management Component:

Execution Time: The multi-threaded approach in the server allowed parallelism allowing multiple clients to be serviced simultaneously, reducing execution, and waiting times for clients. This won't be extremely noticeable on a smaller scale, however, as the number of customers increases (loop iterates more) and the more complex the operations are. In such cases, the CPU is put under higher strain, and the benefits of using threading techniques become clearer. This results in decreased execution time and a smoother user experience during peak loads.

Scalability: As more clients connect to the server, more threads are dynamically created to handle their requests. Each client now has its independent connection with the server, and the threading mechanism enables the CPU to seamlessly switch between these threads quickly, effectively handling them as if they were executing in isolation. This scalable architecture ensures that the system can efficiently accommodate an increasing number of users while maintaining performance.

Memory utilization and synchronization overhead: The decision between having variables within the thread function or globally and synchronizing via mutex is determined by the context and requirements of the software. Both approaches have advantages and downsides, however, I chose to declare the variables within the thread function since it matches with my system objective the best. Here is a breakdown:

1. Performance: Keeping variables local to each thread offers better efficiency especially in my case where variables are often accessed and modified within the thread. Moreover, local variables are often more successfully optimized by the compiler and reduce conflict for shared resources. On the other hand, accessing global variables protected by mutexes may result in some overhead because of the locking and unlocking processes. If many threads often access and modify these global variables, performance may suffer.

2. Efficiency: In this approach, each thread has its own copy of the data it works with, which is easy to keep track of in memory. However, it can use a lot of memory because the same data is stored separately for each thread. On the other hand, using global variables can save memory because they are shared among all threads, but it can become inefficient when multiple threads need to access and change the data frequently. This can lead to conflicts between threads (mutex disputes) which can slow things down and make the program less efficient overall.

3. Parallelism: Since each thread acts separately with its own set of variables, this technique enables parallelism. Parallel execution will also make it easier to debug. The other technique can also allow parallelism, but it necessitates careful mutex management to avoid bottlenecks and deadlocks. When using global variables, it may be more difficult to coordinate several threads.

In my code, all variables in the given serverTH function are thread-local meaning that each thread independently initializes and utilizes its own set of these variables and will not share them with other threads. As a result, no contention or synchronization is necessary. Synchronization would be necessary when many threads access shared resources concurrently (like a global variable) and there is a risk of race situations or data corruption. I found that this is simpler, provides higher performance, lower overhead from synchronization, no competition over shared resources (better scalability), and a lower risk of deadlock. It is the most appropriate approach in my case since threads separately handle data from a socket.

Theoretically, if I wanted to make the variables global for better resource utilization and synchronize, the function would look like this.

```
// Global variables
int sfd;
double principal, interestRate;
int numPayments;
char clientName[256];
char clientLastName[256];
pthread_mutex_t mutex; // Mutex to protect global variables

void * serverTH(void * soc) {
    sfd = (intptr_t)(soc);
    pthread_mutex_lock(&mutex); // Protect with mutex
    int n = read(sfd, clientName, sizeof(clientName));
    n = read(sfd, clientLastName, sizeof(clientLastName));
    n = read(sfd, &principal, sizeof(double));
    n = read(sfd, &interestRate, sizeof(double));
    n = read(sfd, &numPayments, sizeof(int));
    pthread_mutex_unlock(&mutex);
    double result = calculateMortgage(principal, interestRate, numPayments);

    pthread_mutex_lock(&mutex);

    n = write(sfd, &result, sizeof(double));
    FILE *file = fopen("loans.txt", "a");
    if (file != NULL) {
        fprintf(file, "Client: %s %s, Monthly Payment: %.2lf\n", clientName, clientLastName, result);
        fclose(file);
    } else {
        printf("Error opening loans.txt for writing.\n");
    }

    pthread_mutex_unlock(&mutex);

    close(sfd);
}
```

Resource Utilization: Each thread runs independently, making efficient use of CPU resources and preventing idle cores during high client loads.

Load Balancing: The server distributes client requests among threads (one client per thread), avoiding any single thread from becoming overburdened while others remain idle. This dynamic load distribution technique enables a well-balanced workload distribution, which is critical for maintaining responsiveness and stability, particularly when dealing with fluctuating client loads. Once the client is done th thread is joined (destroyed).

Exploring other possible parallelization techniques:

Single Instruction, Multiple Data (SIMD): SIMD executes a single instruction across multiple data elements, which are often stored in vectors or arrays. This method is very effective for applications that require doing the same computation on a huge amount of data, such as multimedia processing, scientific simulations, and picture processing. SIMD processors, such GPUs (Graphics Processing Units), are designed to take advantage of this technology, giving significant performance advantages.

Distributed Computing: It solves complex problems by breaking them down into smaller, manageable tasks that can be executed separately on numerous networked computers or network nodes. Each node in the distributed system works on its own portion of the problem (they work concurrently), and the results are combined to get the final solution. Usually used when the computational requirements exceed the capability of a single computer or when fault tolerance and scalability are required, such as in cloud computing, and big data processing.

Vectorization: It converts scalar operations to vector operations. Like SIMD, it is effective in cases were doing the same computation on a huge amount of data is required. However, it is more general and can be used for a broader range of algorithms and computer languages. Vectorization is commonly performed automatically by modern compilers and processors to increase code performance. Usually used in scientific and numerical computing since it reduces execution time for tasks that process massive data arrays or matrices.

Data Parallelism: It divides a task into smaller sub-tasks and processes these sub-tasks concurrently with numerous processing units. Each processing unit works on a distinct segment of the data, and the results are integrated to produce the result. Often employed in parallel computing frameworks (ex. MapReduce), which breaks big datasets into chunks, process them in parallel, and then aggregate the results. This method is effective for analysis and manipulation of large datasets resulting in scalability and increased performance.

Comparison of thread management with other parallelization techniques:

Ease of implementation: Thread management is generally simple to implement, compared to the rest and is compatible with the existing structure of the banking system. On the other hand, distributed computing requires considerable changes to incorporate networked nodes and data distribution algorithms, and vectorization and SIMD may need rewriting and optimizing certain code sections. Although data parallelism is conceptually simpler, it may necessitate the redesign of data structures, algorithms, etc.

Performance gains: multi-threading significantly reduces execution and waiting times, allowing it to handle several clients concurrently, particularly during peak loads. SIMD and vectorization are specialized to specific algorithms or data structures. Distributed computing, on the other hand, is ideal for scaling over several machines but may introduce complexity that exceeds the constraints of the banking system, making it less suited unless huge scalability is needed in the future. Data parallelism is not appropriate for a highly interactive and concurrent system such as a banking server.

Scalability: Thread management excels at scalability by establishing threads dynamically to handle rising client connections. It enables optimal resource use and load balancing, making it an excellent solution for accommodating many users without sacrificing speed. Distributed computing, like multi-threading, provides scalability but requires networked nodes, which may impose additional complications and overhead. SIMD and vectorization can scale effectively within their specialized domains of data-intensive activities, but they are not adaptable to the system.

Compatibility with Application's Requirements: Thread management, provides a versatile and compatible approach, organically aligning with the system's current structure and easily handling the whole spectrum of requirements (efficiency, fault tolerance, scalability, usability, concurrent access, and continuous service availability) with low customization and architectural changes. On the other hand, SIMD and vectorization are specialized computing processes that lack natural support for concurrent user access, fault tolerance, and user-friendliness. While distributed computing is scalable and fault-tolerant, networking adds complexity and may necessitate additional levels of customization for remote access.

Overall, implementing other parallelization techniques to meet the needs of the system would take more effort, and architectural changes than the more adaptive and straightforward approach using threads.

Although threading is the recommended technique in my scenario. In other cases, it has limitations such as complex synchronization, potential overhead, and limited parallelism. This makes it less suitable for tasks with fine-grained parallelism, resource-constrained environments, or dependencies. On the other hand, SIMD excels in processing large datasets with uniform operations, distributed computing excels in

providing scalability across multiple nodes or machines, vectorization with array-based computations, and data parallelism efficiently handles tasks that can be divided into parallel sub-tasks.

Finalizing

Client-server component description (also mentioned at the beginning of the report)

I will develop a socket-based communication system to facilitate interactions between the server (offers a mortgage calculator service), and the worker. The server will handle client requests concurrently and save their information and monthly payment details to a file. The Client component allows clients to input their personal and mortgage details and receive their monthly payment amount. Once no more clients want to access the server, it executes a program to get statistics (min, max, average).

Running and implementation

I created two separate processes, a client process and a server process using the fork system call. Those processes are independent therefore I used sockets for inter-process communications.

In terms of the client-server model, the system operates over a network, allowing students to interact with the server remotely. Communication between the clients and the server is established through sockets following the TCP protocol. Each client creates a socket on their machine, and the server creates a corresponding socket on its end.

For simulation purposes, I utilized the loopback network interface on the same machine. It is crucial to ensure that the server program is initiated before the client, and both the client and server are configured to connect to the same port number. The server is ensured to run first by making it the parent process, that way the CPU starts executing it before moving onto the client process.

To ensure data integrity, I used TCP, a dependable and connection-oriented communication technology. TCP provides a strong mechanism for ensuring that data received from the client reaches the server without errors, duplication, or loss. It accomplishes this by a series of checks, acknowledgments, and retransmissions if needed. By using TCP as the underlying protocol for data transfer in the system, I can be confident that the monthly payment results, and other essential information would be provided reliably and in the correct order, precisely fitting with the user's requirement for data integrity. This comes at a somewhat higher overhead than UDP, but it assures that the system maintains the highest levels of data consistency.

The server offers concurrent execution of the clients by employing threading. The threads were done by first including the pthreah.h library, then creating the thread helper function (void return type and void arguments), then I filled the code with what I want the server to do (ask for and receive name, receive necessary information, calculate, and display payment and write all information to a file). Then I created the threads after accepting them on the server.

The server is made unstoppable using while by including a while (1) function around the server's listening step.

I made sure to allow multiple clients to access the server at the same time by including a for function around the client's code. I've also made the number of clients not static by allowing the user to specify how many users they want to input each time.

I've also made sure to use the exec function carefully as it stops the execution of the current running process and replaces it with the new program. I have done that by using a fork to create a client and used the exec function to run within this child ensuring the parent (server) is still running even after the statistics are run.

Additionally, the use of forks and pipes is employed within the statistics program. Forks are utilized to create child processes that run concurrently with the main process. Pipes are employed to facilitate communication between these child processes and the main process, enabling data sharing and synchronization. This design choice enhances the system's overall efficiency and supports parallel computation, a critical aspect of meeting user requirements effectively.

I defined and implemented the function outside the main so if the bank decides to calculate the monthly payment using a different equation or modify the process, that this will be easily done and wouldn't affect the client. Enhancing efficiency and providing the user with an interface where no matter what changes, they wouldn't be affected.

To run the client-server component, I have run and compiled the page in shell code. After giving the execute command to the bash file and running it. The user can choose the option to start the mortgage system and is asked to specify the port that the client and server will connect to.

Evaluation of functionality, interface design, and ability to handle concurrent tasks in terms of user requirements.

Functionality:

Task distribution:

1. The parent process starts the server, listens for client connections, and generates threads to handle each one.
2. The child process handles client connection and tasks.
3. Threads are utilized to ensure concurrent execution, which allows several customers to be serviced at the same time.
4. In the statistics program, child processes are created to calculate statistics concurrently.

Tasks are distributed well ensuring efficiency of the system and time execution is kept to the lowest making sure that the user has a positive user experience.

Declaring process id variable and associate the value of the fork function to it.

```
pid_t pid;
pid = fork();
```

Concurrency:

1. Multiple clients can connect to the server at the same time, and each client is handled in a distinct thread.
2. Child processes for statistical calculations run concurrently, ensuring efficient resource utilization.

Pid variables for each function.

```
pid_t avg_pid, max_pid, min_pid;
```

Inter - Process communication:

1. One-way pipes are used to exchange the value of the result to the parent function.
2. Pipes are closed once done ensuring efficient resource management.

Pipes creation

```
int avg_pipe[2], max_pipe[2], min_pipe[2];
```

Example of creating a child, closing the read pipe, and writing onto the socket.

```
max_pid = fork();

if (max_pid == 0) {
    // Child process for maximum calculation
    close(max_pipe[0]); // Close read end of the pipe
    float maxPayment = calculateMax(loans, numLoans);
    write(max_pipe[1], &maxPayment, sizeof(float)); // Write the max payment to the pi
    close(max_pipe[1]);
    exit(0);
} else if (max_pid < 0) {
    perror("Fork failed");
    return 1;
}
```

Reading from pipe to pass it to parent.

```
read(max_pipe[0], &maxPayment, sizeof(float));
```

Closing pipes when done.

```
close(max_pipe[1]);
```

Thread communication:

1. Sockets are used for communication between the server and clients, enabling data flow.
2. I made sure that communication is done efficiently and to close resources when no longer needed.

Thread library:

```
#include <pthread.h>
```

Thread helper function:

```
void * serverTH(void * soc){
             //cast the socket to int (file descriptor)
             int n;
             int sfd = (intptr_t)(soc);

            double principal, interestRate;
            int numPayments;
            char clientName[256];
            char clientLastName[256];

            n = read(sfd, clientName, sizeof(clientName));
            n = read(sfd, clientLastName, sizeof(clientLastName));
            n = read(sfd, &principal, sizeof(double));
            n = read(sfd, &interestRate, sizeof(double));
            n = read(sfd, &numPayments, sizeof(int));

            double result = calculateMortgage(principal, interestRate, numPayments);
            n = write(sfd, &result, sizeof(double));

            // Save client's name and monthly payment to loans.txt
            FILE *file = fopen("loans.txt", "a"); // Open the file in append mode
            if (file != NULL) {
                fprintf(file, "Client: %s %s, Monthly Payment: %.2lf\n", clientName, clientLastName, result);
                fclose(file);
            } else {
                printf("Error opening loans.txt for writing.\n");
            }

            close(sfd);
}
```

Declare thread variable:

```
pthread_t sth;
```

Thread creation:

```
pthread_create(&sth,NULL,serverTH,(void *)(intptr_t)newsockfd);
```

Thread joining:

```
pthread_join(sth,NULL);
```

Socket communication (in my case inter-machine communication):

1. The server sets up a socket, binds it to a port, and listens for each client connection.
2. Clients establish connections by creating a socket to communicate with the server.
3. Data is transmitted between the server and clients over sockets, including client information and mortgage computations.
4. After use, sockets are properly closed to prevent resource leakage.

Server side:
Server creates a socket:

```
int sockfd

sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

Server binds the socket to address and port:

```
bind(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr))
```

Server listens for incoming connections

```
listen(sockfd, 5);
```

Client side:
Client creates a socket:

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

Client connects to server:

```
connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr))
```

Clients sends data to server:

```
n = write(sockfd, clientName, sizeof(clientName));
```

Server receive data from server:

```
n = read(sfd, clientName, sizeof(clientName));
```

Server closes socket:

```
close(sfd);
```

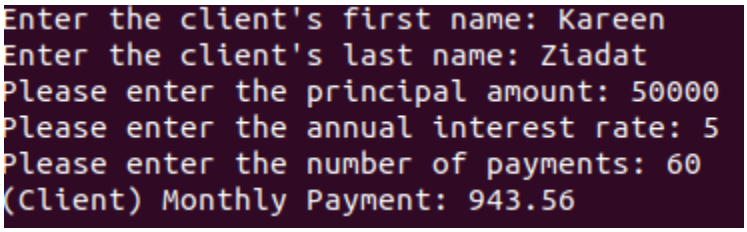
Client closes socket:

```
close(sockfd);
```

Interface design:

User friendliness:

1. The system is text-based and asks for user's names and information through the terminal.
2. The interface relies on users entering their name and numbers into a command-line which is not challenging.
3. Implementing a more user-friendly client interface, such as a GUI or web-based form, could improve user friendliness.



Error handling:

1. It handles errors during socket creation, binding, and connection.
2. It handles errors when opening files.
3. Further error handling can be put in place to better handle exceptions and avoid program termination on errors like when a user enters characters instead of numbers.

References

WhatIs.com. (n.d.). What is process? - Definition from WhatIs.com. [online] Available at: https://www.techtarget.com/whatis/definition/process#:~:text=A%20process%20is%20an%20instance.

Wang, K.C. (2018). Systems Programming in Unix/Linux.

Quora. (n.d.). What is the difference between multitasking and concurrency in programming? [online] Available at: https://www.quora.com/What-is-the-difference-between-multitasking-and-concurrency-in-programming [Accessed 15 Aug. 2023].

4.1 Introduction. (n.d.). Available at: https://www.uobabylon.edu.iq/eprints/publication_3_27434_1357.pdf.

GeeksforGeeks. (2019). Difference between Multi-tasking and Multi-threading. [online] Available at: https://www.geeksforgeeks.org/difference-between-multi-tasking-and-multi-threading/.

SearchDataCenter. (n.d.). What is a Shell Script and How Does it Work? [online] Available at: https://www.techtarget.com/searchdatacenter/definition/shell-script.

BYJUS. (n.d.). Multitasking Operating System | GATE Notes. [online] Available at: https://byjus.com/gate/multitasking-operating-system-notes/.

Stack Overflow. (n.d.). language agnostic - What is the difference between concurrency and parallelism? [online] Available at: https://stackoverflow.com/questions/1050222/what-is-the-difference-between-concurrency-and-parallelism.

www.sciencedirect.com. (n.d.). Single Instruction Multiple Data - an overview | ScienceDirect Topics. [online] Available at: https://www.sciencedirect.com/topics/computer-science/single-instruction-multiple-data.

Storage, P. (2022). Parallel vs. Distributed Computing: An Overview. [online] Pure Storage Blog. Available at: https://blog.purestorage.com/purely-informational/parallel-vs-distributed-computing-an-overview/.

Wikipedia Contributors (2019). Data parallelism. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Data_parallelism.

www.tutorialspoint.com. (n.d.). Operating System - Processes - Tutorialspoint. [online] Available at: https://www.tutorialspoint.com/operating_system/os_processes.htm.

GeeksforGeeks. (2019). Difference between Multi-tasking and Multi-threading. [online] Available at: https://www.geeksforgeeks.org/difference-between-multi-tasking-and-multi-threading/