

# **SISTEMAS OPERATIVOS**

**UNIVERSIDAD FRANCISCO DE VITORIA**

**ESCUELA POLITÉCNICA SUPERIOR**



## **Práctica Final Parte I**

Javier Antón Ordoñez  
Juan Diego Panchón Vidal  
Jaime Ordobás Cubera

<https://github.com/Javiton2005/Sistemas-Operativos>

1. Resumen .....	3
2. Estructuración del código fuente .....	4
3. Explicación de Funciones .....	5
3.1. Main (Banco.c).....	5
3.2. Monitor.c.....	6
3.3. Login.c .....	7
3.4. Crear_Lista_Usuarios/Transacciones .....	8
3.5. Leercsv.c.....	8
3.6 LeerCsvNcuenta .....	8
3.7. Cancelar tarjeta .....	9
3.8. Pagar tasa .....	9
3.9. Transaccion.....	10
3.10. Meter dinero .....	11
3.11 Sacar dinero .....	11
3.12 Menu funciones .....	12
3.13 Info Cuenta.....	13
3.14 Consultar saldo .....	13
3.15 Escribir en el log trans .....	14
3.16 Registrar transacción .....	14
3.17 Editar CSV .....	15
3.18 Escribir en el log.....	15
3.19 Init global.....	16
3.20 Usuario.....	17
4. Limitaciones y problemas encontrados.....	17

# 1. Resumen

Este sistema imita el funcionamiento de un banco real, proporcionando una interfaz de control basada en terminal. Está desarrollado en C, lo que permite un manejo eficiente de la memoria y los procesos del sistema operativo, asegurando un rendimiento óptimo y un alto grado de control sobre las operaciones críticas del sistema.

El sistema se compone de dos programas principales:

- **Banco:** Es el núcleo del sistema, encargado de gestionar las cuentas, procesar transacciones, verificar la autenticidad de los usuarios y administrar la concurrencia en las operaciones financieras.
- **Usuario:** Es la interfaz con la que interactúan los clientes. A través de este programa, los usuarios pueden realizar diversas acciones bancarias mediante comandos en la terminal.

Como se mencionó anteriormente, este sistema está hecho en C para poder controlar a bajo nivel la memoria y los procesos del sistema operativo. Está diseñado para correr en un SO Linux con entorno de escritorio y con mínimo una de las dos terminales instaladas.

- GNOME Terminal
- Kitty terminal

El sistema soporta la gestión de múltiples usuarios simultáneamente, permitiendo incluso que un mismo usuario pueda iniciar sesión desde distintas sesiones sin que esto afecte la integridad de sus datos. Gracias a la implementación de un modelo basado en hilos, cada usuario puede ejecutar sus operaciones de manera independiente, evitando bloqueos o interferencias entre sesiones.

Los usuarios tendrán la posibilidad de realizar distintas acciones bancarias como:

- Sacar dinero
- Meter dinero
- Consultar saldo
- Consultar información de la cuenta
- Pagar tasas
- Cancelar tarjetas
- Transferir dinero

Para garantizar la seguridad y la correcta ejecución de las transacciones, cada operación se ejecuta en un hilo separado. Esto evita problemas como condiciones de carrera o sobreescritura de datos, asegurando que todas las transacciones se procesen de manera segura y eficiente.

El uso de bloqueos y sincronización de hilos se ha tenido muy en cuenta para evitar inconsistencias en los datos cuando múltiples usuarios acceden simultáneamente a la misma cuenta. Esto permite que las operaciones bancarias sean procesadas de manera concurrente sin comprometer la integridad de la información.

## 2. Estructuración del código fuente

El código fuente del sistema está organizado en distintos archivos, cada uno encargado de una función específica. Esta división permite una mejor colaboración entre los miembros del equipo, además de mejorar la modularidad y la abstracción del sistema, facilitando su mantenimiento y escalabilidad.

En la raíz del proyecto se encuentran los ejecutables principales, *banco* y *usuario*, que representan los dos programas del sistema. También está el archivo *Makefile*, que permite compilar el proyecto desde la terminal de manera automatizada, junto con los archivos *usuario.c* y *banco.c*, que contienen las funciones *main* de cada programa. Además, se incluye un fichero de configuración llamado *properties.txt*, donde se almacenan variables ajustables del sistema, como límites de transacciones o tiempos de espera.

Por otro lado, se encuentra una estructura de directorios para el resto de las funciones.

- **Común/:** Contiene un *comun.h* y *init\_global* donde se guardan todas las variables globales, las funciones de editar y leer csv, y escribir en log.
- **Fichero/:** Contiene *banco.log* y *transacciones.log* para el registro de acciones en el banco, *db.csv* para guardar datos de usuarios registrados, y *transacciones.csv* para guardar datos de cada transacción.
- **Funciones/:** Contiene todas las funciones de operaciones bancarias, entre ellas: sacar dinero, meter dinero, consultar saldo, consultar información de la cuenta, pagar tasas, cancelar tarjetas y transferir dinero. También incluye un *funciones.h* para guardar elementos necesarios de funciones.
- **Login/:** Contiene *login.c* para el registro de usuarios existentes o nuevos y su cabecera respectiva, *login.h*.
- **Monitor/:** Contiene *monitor.c* para las identificaciones de anomalías y su respectiva cabecera *monitor.h*.
- **Transacciones/:** Contiene *crear\_transacciones.c* para guardar nuevas transacciones del CSV a una estructura *TRANSACCION*. Tiene *crear\_lista\_transacciones.c* para crear una lista de la estructura transacciones, y un *transacciones.h* como su cabecera.
- **Usuarios/:** Contiene *crear\_usuarios.c* para guardar nuevos usuarios del CSV a una estructura *USUARIO*. Tiene *crear\_lista\_usuarios.c* para crear una lista de la estructura usuarios, y un *usuarios.h* como su cabecera.

Para mantener un código estructurado y claro, las distintas funcionalidades del programa se organizan en carpetas específicas. Por ejemplo, el manejo de transacciones se encuentra en una carpeta dedicada, mientras que la gestión de usuarios se almacena en otra. También existe una carpeta común con funciones compartidas entre ambos programas, como la lectura de archivos o la sincronización de hilos.

## 3. Explicación de Funciones

### 3.1. Main (Banco.c)

Banco.c es el proceso principal que crea una lista de usuarios y llama al proceso de *Login*. Si el usuario introduce el carácter \*, el proceso finaliza.

El programa incluye varias cabeceras, algunas de ellas propias del proyecto como "*comun/comun.h*" o "*usuarios/usuarios.h*" que hacen referencia a *comun.h* y *usuarios.h*.

También se incluyen cabeceras estándar como `<stdio.h>`, `<stdlib.h>`, `<unistd.h>`, `<sys/types.h>`, `<sys/wait.h>`, y `<string.h>` para manejar operaciones de entrada/salida, gestión de memoria, procesos, pipes y manipulación de cadenas.

#### 3.1.1. Void Manejar\_Anomalia (char \*mensaje)

La función *manejar\_anomalia(char \*mensaje)* se encarga de gestionar los mensajes de anomalía que se reciben a través del pipe entre banco y monitor. De entrada, se introduce un mensaje de tipo *char \*mensaje* y este imprime el mensaje completo recibido. A continuación, se extrae el código de la anomalía convirtiendo el carácter a partir del índice 9 a un entero. Esto se debe a que el mensaje tiene un formato fijo en el que el código de la anomalía se encuentra a partir de la posición 9.

A continuación, se utiliza un switch para identificar el tipo de anomalía basado en constantes definidas en el *comun.h*, e imprime un mensaje específico junto con una acción necesaria para manejar esa anomalía en cada caso. Si el código no coincide con ninguna anomalía registrada, el default se encargaría de imprimir "Anomalía desconocida".

#### 3.1.2. Int Main()

La función principal coordina el funcionamiento del banco. Primero, se comienza declarando un puntero doble a *USER* llamado *listaUsuarios*. Después, se define una variable *salir* que controla la salida del bucle principal, y se llama a *InitGlobal()* para inicializar variables globales y configuraciones necesarias.

En cuanto a la creación del pipe. Se crea un array *pipe\_alerta[2]* que contendrá los descriptores de lectura y escritura, y se comprueba si hubo error al crearse. Esto permite la comunicación entre el proceso principal y el monitor, principalmente para comunicar las anomalías detectadas por *Monitor.c*. A continuación, se llama a *fork()* para crear un proceso hijo, comprobando nuevamente si hubo error al hacer *fork()* usando un *if(pid < 0)*. En caso de *if(pid == 0)*, se cierra la lectura (*pipe\_alerta[0]*) y se abre la escritura (*pipe\_alerta[1]*) del pipe. El proceso monitor se encarga de supervisar y generar las alertas de anomalías.

El bucle principal de ejecución consiste en un bucle que se ejecuta mientras la variable *salir* sea distinta de \*. Dentro de él, se genera la lista de usuarios a partir de un archivo de cuentas usando la función *CrearListaUsuarios()* y se verifica si la lista se creó correctamente. En caso de error, se imprime un mensaje y se termina el programa:

En el proceso de login, se llama a la función *login(listaUsuarios)*, que gestiona la autenticación de los usuarios.

A continuación, comienza el proceso de lectura desde el pipe, donde se declara un buffer de 256 caracteres y se entra en un bucle que lee del pipe. Cada vez que se recibe un mensaje, se llama a *manejar\_anomalia(buffer)* para gestionar la alerta.

Para finalizar, una vez que el proceso hijo termina, la función *main()* retorna 1, lo que indica la finalización del programa.

## 3.2. Monitor.c

Aunque monitor.c no funcione correctamente en la solución, este se ejecuta al ser llamado por las funciones de acciones bancarias.

### 3.2.1. Void monitor(int fd\_alerta)

Función principal que gestiona la detección de anomalías y la comunicación con el banco. Recibe un entero que representa el descriptor del pipe para enviar mensajes de anomalías al banco.

1. Asigna el descriptor del pipe para comunicación con el banco.
2. Crea y lanza múltiples hilos para la detección de anomalías.
3. Ejecuta un bucle infinito para notificar a los hilos sobre nuevas transacciones.

No devuelve un valor explícito, ya que su propósito principal es supervisar el proceso de detección de anomalías.

### 3.2.2. Void registrar\_anomalia(int codigo\_anomalia)

Función que registra anomalías detectadas en un archivo de log y envía un mensaje al banco mediante un pipe.

Recibe un entero que representa el código de la anomalía a registrar.

1. Prepara un mensaje con el código de la anomalía utilizando la función *sprintf()*.
2. Registra el mensaje en un archivo de log mediante la función “EscribirEnLog”.
3. Envía el mensaje al banco a través del pipe ``fd_pipe``.

No devuelve un valor explícito, ya que su propósito principal es realizar el registro de la anomalía.

### 3.2.3. Void \*hilo\_fondos\_insuficientes(void \*arg)

Función de un hilo que detecta cuentas con saldo negativo.

Recibe un puntero genérico, aunque en este caso no utiliza parámetros adicionales.

1. Bloquea el acceso a recursos compartidos mediante un mutex.
2. Espera la señal de verificación utilizando una variable de condición.
3. Verifica en la lista de transacciones si alguna cuenta tiene saldo negativo.
4. Si se detecta una anomalía, llama a la función ``registrar_anomalia`` con el código correspondiente.

Siempre retorna NULL porque no se espera ningún valor de retorno.

### 3.2.4. Void \*hilo\_transacciones\_grandes(void \*arg)

Función de un hilo que detecta transacciones con montos superiores a 1000.

Recibe un puntero genérico, aunque en este caso no utiliza parámetros adicionales.

1. Bloquea el acceso a recursos compartidos mediante un mutex.
2. Espera la señal de verificación utilizando una variable de condición.
3. Revisa la lista de transacciones para detectar montos superiores al límite establecido.
4. Si se detecta una anomalía, llama a la función `registrar\_anomalia` con el código correspondiente.

Siempre retorna NULL porque no se espera ningún valor de retorno.

### **3.2.5. Void \*hilo\_secuencia\_inusual(void \*arg)**

Función de un hilo que detecta tres transacciones realizadas por el mismo usuario en menos de un minuto.

Recibe un puntero genérico, aunque en este caso no utiliza parámetros adicionales.

1. Bloquea el acceso a recursos compartidos mediante un mutex.
2. Espera la señal de verificación utilizando una variable de condición.
3. Identifica ventanas de tiempo en las que un usuario realiza tres transacciones en menos de un minuto.
4. Si se detecta una anomalía, llama a la función `registrar\_anomalia` con el código correspondiente.
5. Reinicia la variable de verificación para esperar nuevas señales.

Siempre retorna NULL porque no se espera ningún valor de retorno.

### **3.2.6. Void \*hilo\_usuario\_no\_existe(void \*arg)**

Función de un hilo que verifica transacciones asociadas a usuarios no registrados.

Recibe un puntero genérico, aunque en este caso no utiliza parámetros adicionales.

1. Bloquea el acceso a recursos compartidos mediante un mutex.
2. Espera la señal de verificación utilizando una variable de condición.
3. Revisa si alguna transacción está asociada a un usuario cuya información no está registrada.
4. Si se detecta una anomalía, llama a la función `registrar\_anomalia` con el código correspondiente.

Siempre retorna NULL porque no se espera ningún valor de retorno.

### **3.2.7. Void notificar\_hilos()**

Función que notifica a los hilos sobre la existencia de nuevas transacciones para analizar.

No recibe parámetros explícitos.

1. Bloquea el acceso a recursos compartidos mediante un mutex.
2. Actualiza la variable de verificación para indicar la existencia de nuevas transacciones.
3. Carga las listas de transacciones y usuarios desde archivos de configuración.
4. Desbloquea a todos los hilos mediante una señal de la variable de condición.

No devuelve un valor explícito, ya que su propósito principal es coordinar la comunicación entre los hilos.

## **3.3. Login.c**

Esta función recibe una un doble puntero de usuarios y no devuelve nada.

Lo primero que hace es pedirle las credenciales, usuario y contraseña. Luego ejecuta un bucle con el número de usuarios en la lista y si alguno coincide con el usuario y la contraseña insertada se duplicará el proceso para luego morir ejecutando un `exec`, con el programa Usuario pasando por parámetros el id del usuario que acaba de hacer log in. Tenemos la condición de las dos terminales por si acaso alguna terminal no estuviese instalada. Y llama al fichero escribir en log para registrar el inicio de sesión y, si no coinciden las contraseñas, al final del bucle registra con un aviso (`warning`).

## 3.4. Crear\_Lista\_Usuarios/Transacciones

Existen dos carpetas con los siguientes archivos, con el mismo funcionamiento y flujo del programa para crear listas de usuarios y listas de transacciones.

### 3.4.1. *Transacciones.h / Usuarios.h*

Es un archivo de cabecera donde se declaran las funciones *Crear\_Transaccion/Usuario* y *Crear\_Lista\_Transacciones/Usuarios*, así como la estructura *TRANSACCION/USUARIO*. También incluye una dependencia del archivo *comun.h*.

### 3.4.2. *Crear\_Lista\_Transacciones.c / Crear\_Lista\_Usuarios.c*

Lee un archivo CSV de transacciones/db línea por línea y crea una lista de transacciones/usuarios. Cada línea se convierte en una estructura *TRANSACCION/USUARIO*, que se almacena en una lista dinámica de punteros.

### 3.4.3. *Crear\_Transacciones.c / Crear\_Usuarios.c*

Implementa la función *Crear\_Transaccion/Usuario*, que toma una línea de texto del archivo CSV y la convierte en una estructura *TRANSACCION/USUARIO*. Esta función separa los campos de la línea usando `strsep` para obtener los valores de cada transacción/usuario.

## 3.5. Leercsv.c

### 3.5.1. *USER \*LeerCsv(int \*idLinea)*

Esta función lo que hace es dado un id del usuario se recorre el archivo de cuentas y manda la línea que coincida con el id a la función *CrearUsuario* y retorna el puntero al usuario

## 3.6 LeerCsvNcuenta

La función *LeerCSVNcuenta* busca en un archivo CSV de cuentas un usuario específico en función de su número de cuenta (*ncuenta*). Devuelve un puntero a un entero que indica la posición del usuario en el archivo si lo encuentra, o `NULL` si no lo encuentra.

Recibe un puntero al número de cuenta que se desea buscar en el archivo CSV.

- Devuelve un puntero a un entero (`int *`) que contiene la posición del usuario en el archivo si el número de cuenta existe.
- Devuelve `NULL` si el número de cuenta no se encuentra o si ocurre algún error.

1. Se inicializa un puntero a un entero (*i*) con valor 1 para llevar el conteo de líneas.



2. Se intenta abrir el archivo CSV especificado en Config.archivo\_cuentas.
3. Se lee la primera línea (cabecera del archivo).
4. Se itera por el resto de las líneas para buscar un número de cuenta que coincida con ncuenta.
  - Cada línea se divide en partes: nombre, contraseña y ncuenta.
  - Si el número de cuenta coincide, se cierra el archivo y se retorna la posición encontrada.
5. Si no se encuentra coincidencia, se liberan los recursos y se retorna NULL.

## 3.7. Cancelar tarjeta

### 3.7.1. Void CancelarTarjeta(int \*idUser)

Función que prepara un hilo para actualizar el número de cuenta de un usuario.

Recibe un puntero a un entero que representa el ID del usuario que solicita la cancelación.

1. Solicita al usuario que introduzca el nuevo número de cuenta.
  2. Realiza una verificación inicial:
    - Si el nuevo número de cuenta no es válido (por ejemplo, no es un número), la función finaliza.
  3. Prepara una estructura "IdValor" que contiene:
    - El ID del usuario que realiza la solicitud.
    - El nuevo número de cuenta.
  4. Crea y lanza un hilo para ejecutar la función "CancelarTarjetaHilo", pasando como parámetro la estructura "IdValor".
- No devuelve un valor explícito, ya que su propósito principal es gestionar el hilo.

### 3.7.2. Void CancelarTarjetaHilo(void \*valor)

Función que actualiza el número de cuenta de un usuario.

Recibe un puntero genérico que apunta a una estructura "IdValor", que contiene el ID del usuario y el nuevo número de cuenta.

1. Abre un semáforo para garantizar la sincronización en el acceso al archivo CSV.
  2. Lee los datos del usuario del archivo CSV utilizando el ID proporcionado.
  3. Actualiza el número de cuenta del usuario con el nuevo valor proporcionado.
  4. Guarda los cambios del usuario en el archivo CSV mediante la función "EditarCsv".
  5. Cierra el semáforo para liberar el recurso y permitir el acceso a otros procesos.
- Siempre retorna NULL porque no se espera ningún valor de retorno.

## 3.8. Pagar tasa

### 3.8.1. Void PagarTasas(int \*idUser)

Función principal que prepara un hilo para realiza el pago de tasas de un usuario.

Recibe un puntero a un entero que representa el ID del usuario que realiza el pago.

1. Solicita al usuario la ruta del archivo que contiene la información de la tasa.
2. Verifica si el archivo existe y es válido:
  - Si no existe o no se puede abrir, la función emite un mensaje de error y finaliza.
  - Lee la cantidad desde el archivo y la convierte a formato numérico.
3. Realiza validaciones iniciales sobre la cantidad:
  - Si es negativa, emite un mensaje de formato incorrecto.
  - Si excede el límite establecido por el banco, emite un mensaje de advertencia y finaliza.
4. Prepara una estructura "IdValor" que contiene:

- El ID del usuario.
- La cantidad de la tasa.

5. Crea y lanza un hilo para ejecutar la función “PagarTasasHilo”, pasando como parámetro la estructura preparada.

No devuelve un valor explícito, ya que su propósito principal es lanzar el hilo.

### **3.8.2. Void \*PagarTasasHilo(void \*valor)**

Función que realiza el pago de tasas de un usuario.

Recibe un puntero genérico que apunta a una estructura “IdValor”, que contiene el ID del usuario y la cantidad que se desea pagar.

1. Abre o crea un semáforo para garantizar la sincronización en el acceso al archivo CSV.
2. Obtiene los datos del usuario desde el archivo CSV utilizando el ID proporcionado.
3. Descuenta la cantidad de la tasa del saldo del usuario.
4. Registra la transacción en el archivo de logs:
  - Incluye detalles como la cantidad pagada, la cuenta del usuario y la fecha.
  - La descripción del registro es “Pago de tasa”.
5. Actualiza los datos del usuario en el archivo CSV.
6. Libera el semáforo para permitir el acceso a otros procesos.
7. Llamar a `notificar_hilos()` para verificar posibles anomalías.

Siempre retorna `NULL` porque no se espera ningún valor de retorno.

## **3.9. Transaccion**

### **3.9.1. Void Transaccion(int \*idUser)**

Función que prepara un hilo para realizar una transacción.

Recibe un puntero a un entero que representa el ID del usuario que inicia la transacción.

1. Solicita al usuario la cuenta de destino y la cantidad a transferir.
2. Realiza verificaciones iniciales:
  - Si la cantidad es negativa, informa de un formato incorrecto.
  - Si la cantidad excede el límite establecido, emite un mensaje de advertencia.
3. Prepara una estructura “IdValor” con los datos necesarios para la transacción:
  - ID del usuario actual y la cantidad.
  - ID del usuario objetivo (obtenido del archivo CSV mediante el semáforo).
4. Crea y lanza un hilo que ejecuta la función “TransaccionHilo”, proporcionando la estructura preparada.

No devuelve un valor explícito, ya que finaliza tras crear el hilo.

### **3.9.2. Void \*TransaccionHilo(void \*valor)**

Función que realiza una transacción entre dos usuarios.

Recibe un puntero genérico que apunta a una estructura “IdValor”, que contiene los datos necesarios para realizar la transacción.

1. Abre un semáforo para garantizar la sincronización en el acceso al CSV.
2. Obtiene los datos de los usuarios involucrados en la transacción desde el CSV.
3. Modifica los saldos de los usuarios sumando y restando el valor transferido.
4. Crea un registro de la transacción con detalles como la fecha, las cuentas implicadas y la cantidad transferida.
5. Actualiza la información de los usuarios en el archivo CSV.
6. Libera los recursos y cierra el semáforo.

7. Llamar a `notificar_hilos()` para verificar posibles anomalías.  
Siempre retorna `NULL` porque no devuelve un resultado explícito.

## 3.10. Meter dinero

### 3.9.1. *Void MeterDinero(int \*idUser)*

Función que prepara un hilo para ingresar dinero a la cuenta de un usuario.

Recibe un puntero a un entero que representa el ID del usuario que realiza el ingreso.

1. Solicita al usuario que introduzca la cantidad de dinero a ingresar.
2. Realiza validaciones iniciales:
  - Si la cantidad es negativa o cero, informa al usuario sobre un formato incorrecto y finaliza.
  - Si la cantidad excede el límite de transferencia permitido por el banco, emite un mensaje de advertencia y finaliza.
3. Prepara una estructura "IdValor" que contiene:
  - El ID del usuario.
  - La cantidad de dinero a ingresar.
4. Crea y lanza un hilo que ejecuta la función "MeterDineroHilo", proporcionando como parámetro la estructura preparada.

No devuelve un valor explícito, ya que su principal objetivo es lanzar el hilo.

### 3.9.2. *Void \*MeterDineroHilo(void \*valor)*

Función que ingresa dinero a la cuenta de un usuario.

Recibe un puntero genérico que apunta a una estructura "IdValor", que contiene el ID del usuario y la cantidad a ingresar.

1. Abre o crea un semáforo para garantizar la sincronización en el acceso al archivo CSV.
2. Obtiene los datos del usuario del archivo CSV utilizando el ID proporcionado.
3. Incrementa el saldo del usuario en función de la cantidad especificada.
4. Registra la transacción en el archivo de logs:
  - Detalla la cantidad, la cuenta de origen y la fecha.
  - La descripción de la transacción es "Ingreso manual".
5. Actualiza la información del usuario en el archivo CSV.
6. Libera el recurso del semáforo para permitir acceso concurrente a otros procesos.
7. Llamar a `notificar_hilos()` para verificar posibles anomalías.

Siempre retorna `NULL` porque no se espera un valor de retorno.

## 3.11 Sacar dinero

### 3.11.1. *void \*\_HiloSacarDinero(void \*valor)*

La función "\_HiloSacarDinero" es ejecutada como un hilo independiente. Su propósito principal es procesar un retiro de dinero de un usuario asegurando consistencia en los datos mediante semáforos, verificando que el saldo sea suficiente y registrando las transacciones en el sistema.

Recibe un puntero genérico que apunta a una estructura "IdValor", que contiene los datos necesarios para realizar el ingreso.

Siempre retorna `NULL` porque no se espera un valor de retorno.

Detalles de implementación:

1. Comprueba si el parámetro “valor” es “NULL”. Si es así, imprime un error y finaliza el programa.
2. Utiliza un semáforo para sincronizar el acceso al archivo CSV donde se almacenan los datos del usuario.
3. Recupera la información del usuario con la función “leerCsv”.
4. Verifica si el saldo es suficiente para el retiro. Si no lo es, registra un mensaje de advertencia en el log y finaliza el hilo.
5. Si el saldo es suficiente, actualiza el saldo del usuario.
6. Registra la transacción en el log de transacciones con los detalles relevantes.
7. Actualiza los datos del usuario en el archivo CSV mediante la función “EditarCsv”.
8. Libera los recursos asignados dinámicamente y el semáforo antes de finalizar.
9. Llamar a `notificar_hilos()` para verificar posibles anomalías.

### **3.11.2. void SacarDinero(int \*idUser)**

La función “SacarDinero” solicita al usuario la cantidad de dinero que desea retirar, realiza validaciones básicas y crea un hilo para ejecutar la función “\_HiloSacarDinero”.

Recibe un puntero al identificador del usuario (int \*) que realiza la operación de retiro.

Valor de retorno:

- No tiene valor de retorno.

Detalles de implementación:

1. Solicita al usuario ingresar la cantidad de dinero que desea retirar.
2. Valida que la cantidad ingresada sea un valor positivo. Si no lo es, imprime un mensaje de error y finaliza la función.
3. Asigna dinámicamente memoria para una estructura “IdValor” que contiene el ID del usuario y la cantidad a retirar.
4. Crea un hilo que ejecuta la función “\_HiloSacarDinero”, pasándole la estructura “IdValor” como parámetro.

## **3.12 Menu funciones**

### **3.12.1. void MenuOpciones(int \*idUser)**

Función que muestra al usuario un menú de opciones y ejecuta la acción correspondiente según la selección realizada.

Recibe un puntero a un entero que representa el ID del usuario que interactúa con el menú.

1. Comprueba si el parámetro “idUser” es NULL. Si lo es, muestra un mensaje de error y termina el programa.
2. Limpia la pantalla utilizando “system(“clear”)”.
3. Recorre el array “funcionesMenu” para imprimir las opciones disponibles del menú, numeradas desde el 1.
4. Solicita al usuario que introduzca una opción utilizando “scanf”.
5. Dependiendo de la opción seleccionada, se realiza una de las siguientes acciones:
  - Opción 1: Ejecuta la función “SacarDinero” con el ID de usuario proporcionado.
  - Opción 2: Ejecuta la función “MeterDinero” con el ID de usuario proporcionado.
  - Opción 3: Ejecuta la función “ConsultarSaldo” con el ID de usuario proporcionado.

- Opción 4: Ejecuta la función “InfoCuenta” con el ID de usuario proporcionado.
- Opción 5: Ejecuta la función “Transaccion” con el ID de usuario proporcionado.
- Opción 6: Ejecuta la función “PagarTasas” con el ID de usuario proporcionado.
- Opción 7: Ejecuta la función “CancelarTarjeta” con el ID de usuario proporcionado.
- Opción 8: Desenlaza los semáforos utilizados en el sistema y termina la ejecución del programa.

6. Si la opción seleccionada no coincide con ninguna de las anteriores, no se realiza ninguna acción.

No devuelve un valor explícito, ya que su propósito principal es ofrecer las opciones del menú y gestionar la ejecución de las acciones seleccionadas.

## 3.13 Info Cuenta

### 3.13.1. *Void InfoCuenta(int \*idUser)*

Función que muestra información detallada de la cuenta de un usuario en pantalla.

Recibe un puntero a un entero que representa el ID del usuario cuya información se desea consultar.

1. Comprueba si el parámetro “idUser” es inválido (menor a 0). Si lo es, la función termina inmediatamente sin realizar ninguna operación.
2. Utiliza la función “leerCsv” para obtener los datos del usuario asociado al ID proporcionado.
3. Limpia la pantalla utilizando “system(“clear”)”.
4. Muestra en pantalla la siguiente información sobre el usuario:
  - Nombre.
  - Contraseña.
  - Número de cuenta.
  - Saldo disponible.
  - Identificador del usuario.
5. Solicita al usuario presionar una tecla para salir, utilizando “getchar” para pausar la ejecución hasta que se reciba una entrada.

No devuelve un valor explícito, ya que su propósito principal es mostrar información en pantalla.

## 3.14 Consultar saldo

### 3.14.1. *Void ConsultarSaldo(int \*idUser)*

Función que muestra el saldo actual de la cuenta de un usuario en pantalla.

Recibe un puntero a un entero que representa el ID del usuario cuya información se desea consultar.

1. Comprueba si el parámetro “idUser” es ‘NULL’. Si lo es, muestra un mensaje de error utilizando “perror” y termina la ejecución del programa con “exit(-1)”.
2. Utiliza la función “leerCsv” para obtener los datos del usuario asociado al ID proporcionado.
3. Limpia la pantalla utilizando “system(“clear”)”.
4. Muestra en pantalla el nombre del usuario y el saldo actual de la cuenta.

5. Solicita al usuario presionar una tecla para salir, utilizando “getchar” para pausar la ejecución hasta que se reciba una entrada.

No devuelve un valor explícito, ya que su propósito principal es mostrar información en pantalla.

## 3.15 Escribir en el log trans

### 3.11.1. *Void EscribirLogTrans(TRANSACCION \*trans)*

Función que registra en un archivo de log la información de una transacción realizada por un usuario.

Recibe un puntero a una estructura “TRANSACCION”, que contiene los detalles de la operación.

1. Obtiene la fecha y hora actual del sistema en formato “dd-mm-aaaa hh:mm:ss”.
2. Prepara el mensaje a escribir en el log con la información de la transacción:
  - Número de cuenta saliente (trans->ncuentas).
  - Número de cuenta objetivo (trans->ncuentao o NULL si no existe en la operación).
  - Cantidad de la transacción (trans->cantidad).
  - Descripción de la transacción (trans->descripcion).
3. Abre un semáforo para garantizar sincronización en el acceso al archivo de log de transacciones.
4. Abre el archivo de log de transacciones especificado en “Config.archivo\_log\_transferencias” en modo de adición.
  - Si el archivo no puede abrirse, libera el semáforo y la memoria asignada, y finaliza la función.
5. Escribe el mensaje de log en el archivo.
6. Cierra el archivo y libera el semáforo.
7. Libera los recursos dinámicamente asignados en la estructura “TRANSACCION”

No devuelve un valor explícito, ya que su propósito principal es registrar la transacción en el archivo de log.

## 3.16 Registrar transacción

### 3.16.1. *Void RegistrarTransaccion(TRANSACCION \*transaccion)*

Función que registra los detalles de una transacción en un archivo específico de transferencias.

Recibe un puntero a una estructura `TRANSACCION` que contiene los datos de la operación.

1. Obtiene la fecha y hora actual del sistema en formato “dd-mm-aaaa hh:mm:ss” utilizando las funciones de tiempo.
2. Verifica si el parámetro “transacción” es NULL. Si lo es:
  - Muestra un mensaje de error en la salida de error estándar (stderr).
  - Termina la función sin realizar ninguna operación.
3. Abre el archivo de transferencias especificado en “Config.archivo\_tranferencias” en modo de adición (“a”).
  - Si el archivo no puede abrirse:
    - Muestra un mensaje de error con “perror”.

- Termina la función.
4. Escribe los detalles de la transacción en el archivo en el siguiente formato:
    - Cantidad transferida (transaccion->cantidad).
    - Número de cuenta saliente (transaccion->ncuentas).
    - Número de cuenta objetivo (transaccion->ncuenta).
    - Fecha y hora de la transacción (fecha).
  5. Cierra el archivo tras escribir los datos.

No devuelve un valor explícito, ya que su propósito principal es registrar la transacción en el archivo de transferencias.

## 3.17 Editar CSV

### 3.17.1. **Void \*EditarCsv(USER \*usuario)**

Función que edita la información de un usuario específico en un archivo CSV, reemplazando la línea correspondiente al usuario con los datos actualizados.

Recibe un puntero a una estructura `USER` que contiene la información del usuario que se desea modificar.

1. Comprueba si el parámetro “usuario” es NULL. Si lo es:
  - Muestra un mensaje de error utilizando “perror”.
  - Finaliza el programa con “exit(EXIT\_FAILURE)”.
2. Abre el archivo original, especificado en “Config.archivo\_cuentas”, en modo de lectura binaria (rb).
  - Si el archivo no puede ser abierto:
    - Muestra un mensaje de error con “perror”.
    - Finaliza el programa con “exit(EXIT\_FAILURE)”.
3. Crea un archivo temporal llamado “temp.csv” en modo de escritura binaria (wb).
  - Si el archivo temporal no puede ser creado:
    - Muestra un mensaje de error con “perror”.
    - Cierra el archivo original y finaliza el programa con “exit(EXIT\_FAILURE)”.
4. Lee línea por línea el archivo original utilizando “fgets” y lleva un contador para identificar la línea correspondiente al usuario proporcionado.
  - Si la línea actual coincide con el ID del usuario (usuario->id), escribe en el archivo temporal los datos del usuario en el formato: “nombre;contraseña;número de cuenta;saldo”.
  - Si no coincide, copia la línea original al archivo temporal.
5. Cierra ambos archivos (el original y el temporal) tras completar la lectura y escritura.
6. Reemplaza el archivo original con el archivo temporal:
  - Elimina el archivo original utilizando “remove”.
  - Renombra el archivo temporal a la ubicación del archivo original utilizando “rename”.
7. Retorna NULL al finalizar, ya que no se espera un valor explícito de retorno.

## 3.18 Escribir en el log

### 3.18.1. **Void EscribirEnLog(char \*frase)**

Función que registra un mensaje en el archivo de log, incluyendo la fecha y hora del sistema para cada entrada.

Recibe una cadena de caracteres (“char \*frase”) que contiene el mensaje que se desea registrar.

1. Calcula la longitud del mensaje, dejando espacio adicional para incluir la fecha y el formato necesario.
2. Obtiene la fecha y hora actual del sistema en formato “dd-mm-aaaa hh:mm:ss” utilizando las funciones de tiempo (“strftime”).
3. Construye el mensaje final que será escrito en el log, concatenando la fecha con el contenido de “frase”.
4. Abre un semáforo con el nombre “/semaforo\_log” para garantizar sincronización en el acceso al archivo de log.
  - Si el semáforo no puede ser abierto, muestra un mensaje de error utilizando “perror” y termina la función.
5. Espera al semáforo (“sem\_wait”) para garantizar que solo un proceso acceda al archivo de log en un momento dado.
6. Abre el archivo de log especificado en “Config.archivo\_log” en modo de adición (“a”).
  - Si el archivo no puede ser abierto, muestra un mensaje de error utilizando “perror”, libera el semáforo y termina la función.
7. Escribe el mensaje en el archivo utilizando “fputs”.
8. Cierra el archivo tras completar la escritura.
9. Libera el semáforo (“sem\_post”) para permitir el acceso a otros procesos y cierra el semáforo (“sem\_close”).
10. Finaliza sin devolver ningún valor explícito, ya que su propósito principal es registrar el mensaje en el archivo.

## 3.19 Init global

### 3.12.3. Void InitGlobal()

Función que inicializa las variables globales “Estadísticas” y “Config”, y configura los valores del sistema leyendo un archivo de propiedades.

1. Inicializa el atributo “usuarios” de la estructura “Estadísticas” a 0.
2. Abre el archivo “properties.txt” en modo de lectura binaria (“rb”) para cargar las configuraciones globales del sistema.
  - Si el archivo no puede ser abierto:
    - Muestra un mensaje de error utilizando “perror”.
    - Finaliza la función.
3. Desenlaza los semáforos existentes relacionados con el sistema para garantizar que no haya conflictos en futuras operaciones:
  - “/semaforo\_dbcsv”.
  - “/semaforo\_log”.
  - “/sem\_log\_trans”.
4. Lee línea por línea el archivo de propiedades. Para cada línea:
  - Ignora comentarios (líneas que comienzan con “#”) y líneas de menos de 3 caracteres.
  - Identifica las claves de configuración y almacena sus valores en la estructura “Config” utilizando “sscanf”:
    - “LIMITE\_RETIRO”: Valor máximo permitido para retiros.
    - “LIMITE\_TRANSFERENCIA”: Valor máximo permitido para transferencias.



- “UMBRAL\_RETIROS”: Umbral de retiros definido por el sistema.
- “UMBRAL\_TRANSFERENCIAS”: Umbral de transferencias definido por el sistema.
- “ARCHIVO\_LOG\_TRANSFERENCIAS”: Ruta del archivo de log de transferencias.
- “ARCHIVO\_CUENTAS”: Ruta del archivo CSV que almacena la información de las cuentas.
- “ARCHIVO\_LOG”: Ruta del archivo general de log.
- “LIMITE\_INTENTOS\_LOGIN”: Número máximo de intentos de inicio de sesión permitidos.
- “ARCHIVO\_TRANSFERENCIAS”: Ruta del archivo donde se registran las transferencias.

5. Cierra el archivo de propiedades tras procesar todas las líneas.

No devuelve un valor explícito, ya que su propósito principal es configurar las variables globales del sistema.

## 3.20 Usuario

### 3.20.1. *Int main(int argc, char \*argv[])*

Función principal que inicia la ejecución del programa, configura las variables globales y permite al usuario interactuar con un menú de opciones.

Recibe los parámetros de la línea de comandos y utiliza un bucle infinito para gestionar continuamente las acciones del usuario.

1. Convierte el argumento de la línea de comandos “argv[1]” (el ID del usuario) a un entero y lo almacena en la variable “idUser”.

- Si no se proporciona un argumento o este no es válido, el valor de `idUser` será 0, lo que puede generar comportamiento inesperado en las funciones subsecuentes.

2. Llama a la función “InitGlobal” para inicializar las variables globales y configurar los valores del sistema.

3. Entra en un bucle infinito (while (1)), donde se llama continuamente a la función “MenuOpciones” pasando el puntero a “idUser” como parámetro.

- Esto permite que el usuario interactúe de forma iterativa con el menú hasta que seleccione una opción que detenga la ejecución del programa (por ejemplo, salir del menú).

No devuelve un valor explícito, ya que esta función está definida como el punto de entrada del programa y no suele retornar un valor en condiciones normales.

## 4. Limitaciones y problemas encontrados

Las limitaciones encontradas en este proyecto han sido nuestro propio conocimiento del lenguaje de programación C.

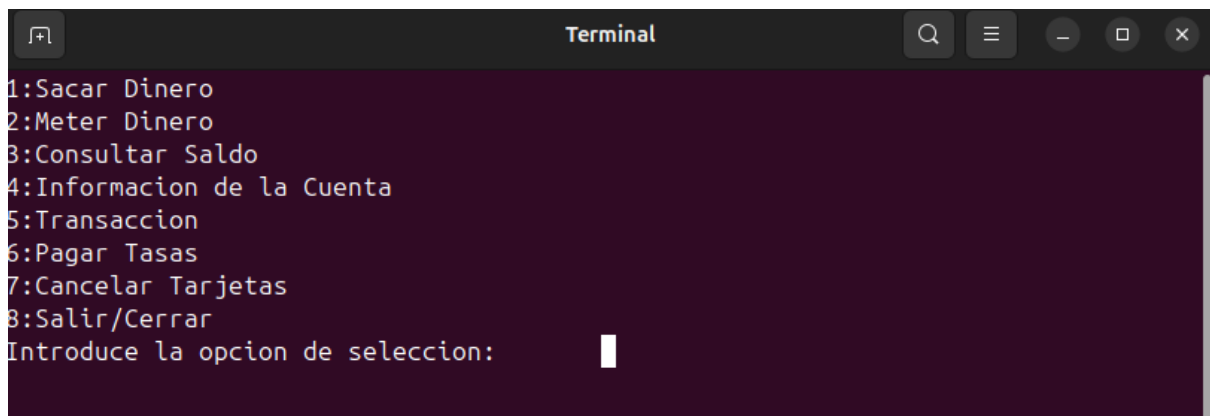
Es un lenguaje de programación en el que hay que controlar la memoria a muy bajo nivel, dando pie a muchos problemas que en un comienzo parecen inexplicables.

Otro problema encontrado es la comunicación entre procesos, más específicamente la comunicación entre el proceso padre e hijo tras el ‘fork’ de banco. Debido a la poca claridad de estructura inicial del trabajo, hubo que pasar por un proceso de prueba y error, hasta encontrar la forma más óptima de realizar el fork() y comunicar ambos procesos.

## 5. Interfaces

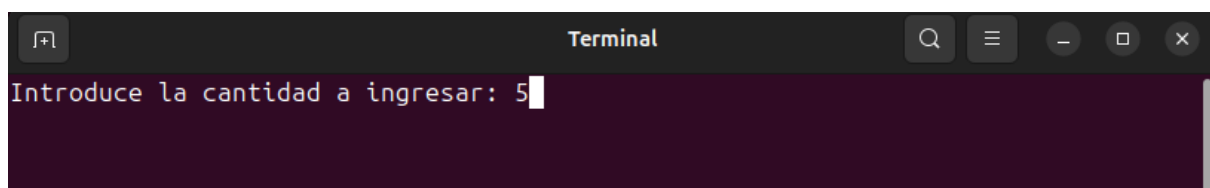
```
> ./banco
Inserta Nombre de la cuenta: javier
Inserta Contraseña de la cuenta: 12345
Presione \n o para salir presiona *: 
```

Al ejecutar Make y ./banco se muestra en pantalla una interfaz de login de usuario y contraseña, donde se deberán introducir datos que estén guardados en la base de datos de usuarios.

A terminal window titled "Terminal" with a dark background. It displays a menu of banking options numbered 1 to 8: 1:Sacar Dinero, 2:Meter Dinero, 3:Consultar Saldo, 4:Informacion de la Cuenta, 5:Transaccion, 6:Pagar Tasas, 7:Cancelar Tarjetas, 8:Salir/Cerrar. Below the menu, it prompts "Introduce la opcion de seleccion:" followed by a cursor.

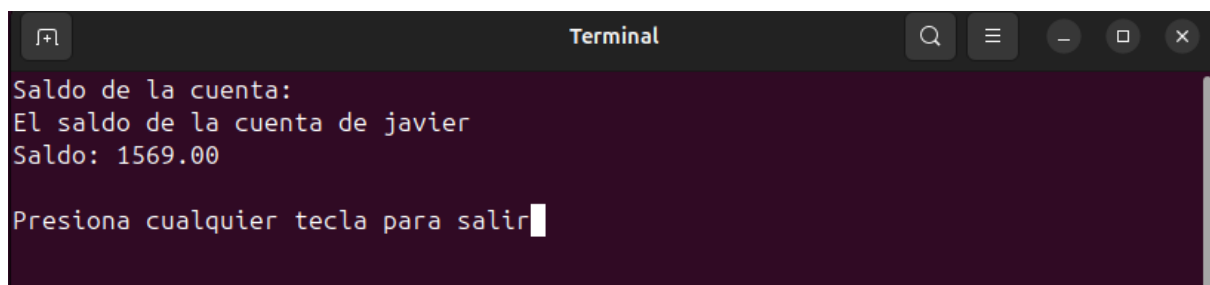
```
Terminal
1:Sacar Dinero
2:Meter Dinero
3:Consultar Saldo
4:Informacion de la Cuenta
5:Transaccion
6:Pagar Tasas
7:Cancelar Tarjetas
8:Salir/Cerrar
Introduce la opcion de seleccion: 
```

Se mostrara el menú de usuario con todas sus posibles acciones bancarias.

A terminal window titled "Terminal" with a dark background. It prompts "Introduce la cantidad a ingresar:" followed by the number 5 and a cursor.

```
Terminal
Introduce la cantidad a ingresar: 5
```

En la opción meter dinero le muestra al usuario la posibilidad de insertar la cantidad a ingresar

A terminal window titled "Terminal" with a dark background. It displays the account balance: "Saldo de la cuenta:", "El saldo de la cuenta de javier", and "Saldo: 1569.00". Below this, it prompts "Presiona cualquier tecla para salir" followed by a cursor.

```
Terminal
Saldo de la cuenta:
El saldo de la cuenta de javier
Saldo: 1569.00

Presiona cualquier tecla para salir
```

Una vez se saque dinero se muestra el saldo actualizado.

```
Terminal
Info de la cuenta:
Nombre: javier
Contraseña: 12345
Numero de cuenta: 611405427
Saldo: 1569.00
Id: 1
Presiona cualquier tecla para salir
```

En la información de la cuenta aparecerán los diversos campos de información que se almacenan del usuario.

```
Terminal
Nº de cuenta al que se le quiera hacer la transacción: 123456789
¿Que cantidad quiere transferir? 5
```

Para las transacciones será necesario introducir un número de cuenta y cantidad valido

```
Ruta del archivo de la tasa:
/tasa.txt
```

Para el pago de tasas será necesario introducir una ruta con un archivo en el que aparezca solo un número.

```
Terminal
Ingrese el nuevo numero de cuenta que quiera tener(9 caracteres): 454545454
```

Para el cambio de numero de cuenta será necesario introducir un numero de 9 dígitos entero.

```
Inserta Nombre de la cuenta: javier
Inserta Contraseña de la cuenta: 12345
Presione \n o para salir presiona *: *
panch@panch-virtual:~/Sistemas-Operativos/lab6$
```

En caso de introducir '\*', el programa se finalizara automáticamente.