Universidad Francisco de Vitoria

Escuela Politécnica Superior

Grado en Ingeniería Informática

Práctica Final Parte II

Trabajo presentado por:	Juan Diego Panchón, Javier Anton, Jaime Ordovás
Profesor/a:	Marlon Cárdenas
Fecha:	

Tabla de contenidos

1. Resumen 5

2. Estructuración del código fuente 7

3. Explicación de Funciones 9

3.1. Main(Banco.c) 9

3.1.1 Int Main() 9

3.2 Monitor (Monitor.c) 9

3.3. Login.c 15

3.3. Crear_Lista_Usuarios/Transacciones 16

3.3.1. Transacciones.h / Usuarios.h 16

3.3.2. Crear_Lista_Usuarios.c 16

3.3.3. Crear_Transacciones.c / Crear_Usuarios.c 16

3.4. Usuario.c 16

3.4.1. main 16

Parámetros: 16

Flujo del programa: 17

3.4.2 CancelarTarjetaHilo 17

Parámetros: 17

Flujo del programa: 17

3.4.3 CancelarTarjeta 17

Parámetros: 17

Flujo del programa: 17

3.4.4 ConsultarSaldo 18

Parámetros: 18

Flujo del programa: 18
3.4.5 InfoCuenta 18
Parámetros:..... 18
Flujo del programa: 18
3.4.6 Menú opciones 18
Parámetros:..... 19
Flujo del programa: 19
3.4.7 MeterDineroHilo..... 19
Parámetros:..... 19
Flujo del programa: 19
3.4.8 MeterDinero..... 20
Parámetros:..... 20
Flujo del programa: 20
3.4.9 RegistrarTransaccion..... 20
Parámetros:..... 20
Flujo del programa: 20
3.4.10 _HiloSacarDinero 20
Parámetros:..... 21
Flujo del programa: 21
3.4.11 SacarDinero..... 21
Parámetros:..... 21
Flujo del programa: 21
3.4.12 TransaccionHilo 21
Parámetros:..... 21
Flujo del programa: 22

3.4.13 Transaccion 22
Parámetros: 22
Flujo del programa: 22
3.5 Común 22
3.5.1 EditarCsv 23
Parámetros: 23
Flujo del programa: 23
3.5.2 EscribirEnLog 23
Parámetros: 23
Flujo del programa: 23
3.5.3 InitGlobal 24
Parámetros: 24
Flujo del programa: 24
3.5.4 LeerCSVNcuenta 24
Parámetros: 24
Flujo del programa: 24
3.5.5 leerCsv 25
Parámetros: 25
Flujo del programa: 25
4. Estructura de ficheros. 26
5. Limitaciones & Problemas Encontrados 27
Conclusión 28

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 tres 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.
- Monitor**:** Es la interfaz en donde se muestra anomalías en las transacciones, mostrando códigos de error dependiendo de la anomalía detectada

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 escritor 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:

1. Sacar dinero
2. Meter dinero
3. Consultar saldo
4. Consultar información de la cuenta
5. Cancelar tarjetas
6. 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. También tendrá una memoria compartida asignada a las sesiones de los usuarios y al proceso principal, banco, para así tener siempre la información actualizada en todas las sesiones.

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 el modularidad y la abstracción del sistema, facilitando su mantenimiento y escalabilidad.

En la raíz del proyecto se encuentran los ejecutables principales, banco, monitor y usuario, que representan los tres 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, monitor.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 , 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, 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.
- 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 en una memoria compartida, junto a su llave 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" "login/login.h"* que hacen referencia a comun.h login.h y usuarios.h.

3.1.1 Int Main()

La función principal coordina el funcionamiento del banco. Primero comienza declarando la llave de para la memoria compartida y llama a la función init global que inicia las variables de entorno. Crea una tabla de usuarios para la memoria compartida con un tamaño de 0 y luego se realocara en la función lista de usuarios.

Después procede a dividirse en un fork para ejecutar el proceso monitor sacrificando uno de los procesos y el otro un bucle del que se puede salir si el usuario inserta * y si no pues manda a la función log in continuamente.

AL final de la función libera la memoria global.

3.2 Monitor (Monitor.c)

Funciones Definidas en monitor.c (Análisis Detallado)

1. void registrar_anomalia(int codigo_anomalia)

   Responsabilidad: Registrar la detección de una anomalía.

   Detalles:

   - Toma un entero codigo_anomalia que identifica el tipo de anomalía.
   - Formatea un mensaje de registro: sprintf(mensaje,"ANOMALÍA %d\n", codigo_anomalia);

- Llama a EscribirEnLog(mensaje); Importante: La *implementación* de EscribirEnLog *no está en este fichero*. Se asume que está definida en comun/comun.h. Desde la perspectiva de monitor.c, EscribirEnLog es una función externa que recibe una cadena y la escribe en el log.
- Imprime un mensaje a la consola: printf("ANOMALÍA n:%d\n", codigo_anomalia); Esto podría ser para propósitos de depuración o información en tiempo real.

2. void liberarMemoria()

   Responsabilidad: Liberar la memoria dinámica asignada dentro de monitor.c.

   Detalles:

   - free(listaUsuarios);: Libera la memoria del array que lleva la cuenta de los logins.
   - free(posiciones);: Libera la memoria del array que guarda las posiciones de lectura en los archivos de transacciones.
   - El bucle for libera la memoria asignada para los arrays internos de ultimasTransacciones:
     - free(ultimasTransacciones[i]);: Libera el array de 3 struct tm para cada usuario.
   - Importante: Esta función es crucial para evitar fugas de memoria, pero su correcta llamada depende de la lógica de control del programa (principalmente en main()).

3. void *hilo_transacciones_grandes(void *arg)

   Responsabilidad: La lógica para el hilo que busca transacciones que exceden límites.

   Detalles:

   - El bucle for itera sobre los usuarios.
   - Construye el nombre del archivo de transacciones: sprintf(nombre_archivo, "./ficheros/%d/transacciones.log", 1001 + i); Esto

asume una estructura de directorios específica (./ficheros/%d/) y una convención de nombres de archivo.

- fopen(nombre_archivo, "r");: Abre el archivo para lectura. El manejo de errores (if(archivo==NULL)) es local a esta función; si un archivo no se abre, el hilo *continúa* con el siguiente usuario.
- fseek(archivo, posiciones[i] , SEEK_SET);: Se posiciona en la última posición leída. posiciones es un array global gestionado por otros hilos (especialmente hilo_comprobar_login y hilo_comprobar_logout).
- El bucle while (fgets(linea, 256, archivo)) lee el archivo línea por línea.
- crearTransaccion(linea);: Importante: La *implementación* de crearTransaccion *no está en este fichero*. Se asume que está en transacciones/transacciones.h. Desde la perspectiva de monitor.c, crearTransaccion es una función externa que toma una línea de texto y devuelve una estructura TRANSACCION.
- La lógica de detección de anomalías: if(transaccion->cantidad>Config.limite_transferencia && transaccion->cantidad>Config.limite_retiro) Aquí, Config *no está definido en este fichero*. Se asume que es una variable global o una estructura accesible externamente.
- Actualiza ultimasTransacciones: Esto asume que ultimasTransacciones es una estructura de datos compartida y que las fechas de las transacciones son importantes para otros hilos.
- free(transaccion); Libera la memoria asignada por crearTransaccion.
- posiciones[i]=ftell(archivo); Actualiza la posición de lectura.
- fclose(archivo); Cierra el archivo.
- pthread_mutex_lock(&mutex); y pthread_mutex_unlock(&mutex); Protegen el acceso a los recursos compartidos *dentro de este hilo*.

4. bool es_fecha_valida(struct tm *fecha)

Responsabilidad: Comprobar si una estructura tm representa una fecha válida (no inicializada).

Detalles:

- Es una función de utilidad pequeña y autocontenida.
- La "validez" se define por si todos los campos de fecha son cero. Esto es una convención específica del programa.

5. void *hilo_secuencia_inusual(void *arg)

   Responsabilidad: La lógica del hilo que detecta secuencias de transacciones demasiado rápidas.

   Detalles:

   - El bucle for itera sobre los usuarios.
   - !es_fecha_valida(&ultimasTransacciones[i][0]) || !es_fecha_valida(&ultimasTransacciones[i][2]): Utiliza la función definida en este fichero para comprobar la validez de las fechas.
   - mktime(&ultimasTransacciones[i][0]); y mktime(&ultimasTransacciones[i][2]);: Convierte las estructuras tm a time_t. mktime es una función de la biblioteca estándar.
   - difftime(t3, t1);: Calcula la diferencia de tiempo. difftime es una función de la biblioteca estándar.
   - La lógica de detección de la anomalía: if (diferencia < 60.0)
   - memset(&ultimasTransacciones[i][j], 0, sizeof(struct tm));: "Limpia" las fechas. Esto es importante porque, de lo contrario, la misma secuencia podría detectarse repetidamente.
   - registrar_anomalia(ESTADO_SECUENCIA_INUSUAL);: Registra la anomalía utilizando la función definida en este fichero.

6. void *hilo_comprobar_login(void *arg)

   Responsabilidad: La lógica del hilo que escucha los inicios de sesión.

Detalles:

- Recibe el descriptor del FIFO de inicio de sesión (fd_lectura_inicio) como argumento.
- El bucle while (!detener_hilos_logs) es el bucle principal del hilo. detener_hilos_logs es una variable global en monitor.c que se usa para señalar al hilo que debe terminar.
- FD_ZERO(&readfds); FD_SET(*fd_lectura_inicio, &readfds);: Configura el conjunto de descriptores de archivo para select().
- timeout.tv_sec = TIMEOUT_SEC; timeout.tv_usec = 0;: Establece el tiempo de espera para select(). TIMEOUT_SEC está definido como 1 en este fichero.
- select(*fd_lectura_inicio + 1, &readfds, NULL, NULL, &timeout); Espera a que haya datos disponibles en el FIFO. select() es una función de la biblioteca estándar.
- read(*fd_lectura_inicio, &numero_user, sizeof(int)); Lee el ID del usuario del FIFO.
- La lógica para manejar el crecimiento dinámico de los arrays listaUsuarios, posiciones, y ultimasTransacciones usando realloc(). Esto es *muy importante* para la escalabilidad del sistema, pero también implica un manejo cuidadoso de la memoria.
- La lógica para leer información de un archivo de configuración (./ficheros/%d/config.conf). El archivo de configuración contiene la posición del archivo de transacciones para el usuario.
- listaUsuarios[index]++;I ncrementa el contador de inicios de sesión.
- Manejo de errores de read() y select().

7. void *hilo_comprobar_logout(void *arg)

   Responsabilidad: La lógica del hilo que escucha los cierres de sesión.

   Detalles:

- Es *muy similar* a hilo_comprobar_login().
- La principal diferencia es que *decrementa* el contador de inicios de sesión (listaUsuarios[index]--;) y *actualiza* el archivo de configuración con la nueva posición del archivo de transacciones.

8. void hilos(pthread_t *hilo_transacciones, pthread_t *hilo_secuencia)

   Responsabilidad: Crear y esperar a que terminen los hilos de detección de anomalías.

   Detalles:

   - pthread_create(hilo_transacciones, NULL, hilo_transacciones_grandes, NULL): Crea el hilo para transacciones grandes. pthread_create es una función de la biblioteca pthreads.
   - pthread_create(hilo_secuencia, NULL, hilo_secuencia_inusual, NULL): Crea el hilo para secuencias inusuales.
   - pthread_join(*hilo_transacciones, NULL): Espera a que el hilo de transacciones grandes termine. pthread_join es una función de la biblioteca pthreads.
   - pthread_join(*hilo_secuencia, NULL); Espera a que el hilo de secuencias inusuales termine.
   - Importante: Esta función *no inicia* los hilos de login/logout. Esos hilos se inician en main().

9. int main(int argc, char *argv[])

   Responsabilidad: La función principal del programa monitor.

   Detalles:

   - InitGlobal(); Importante: La *implementación* de InitGlobal *no está en este fichero*. Se asume que está en comun/comun.h. Es una función externa que inicializa la configuración global del programa.
   - Inicializa variables globales: nusers = 0; pthread_mutex_init(&mutex, NULL); asigna memoria inicial para listaUsuarios y ultimasTransacciones.

- Declara variables para los hilos y los descriptores de archivo de los FIFOs.
- mkfifo(FIFO_INICIO, 0666); y mkfifo(FIFO_CERRAR, 0666): Crea los FIFOs. mkfifo es una función de la biblioteca estándar. El manejo de errores verifica si el FIFO ya existe (errno != EEXIST).
- open(FIFO_INICIO, O_RDONLY | O_NONBLOCK); y open(FIFO_CERRAR, O_RDONLY | O_NONBLOCK): Abre los FIFOs para lectura en modo no bloqueante. O_NONBLOCK es crucial para usar select().
- pthread_create(&hilo_login, NULL, hilo_comprobar_login, &fd_lectura_inicio); y pthread_create(&hilo_cerrar, NULL, hilo_comprobar_logout, &fd_lectura_cerrar) Crea los hilos de login/logout.
- El bucle while (!detener_hilos_logs) es el bucle principal del programa.
- hilos(&hilo_transacciones, &hilo_secuencia); Llama a la función definida en este fichero para crear y esperar a los hilos de anomalías.
- pthread_join(hilo_login, NULL); y pthread_join(hilo_cerrar, NULL): Espera a que los hilos de login/logout terminen.
- return 0: Indica éxito.

3.3. Login.c

Esta función recibe un puntero a la tabla de usuarios de memoria compartida, la id de la memoria compartida 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).

3.3. 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.

3.3.1. Transacciones.h / Usuarios.h

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

3.3.2. Crear_Lista_Usuarios.c

Lee un archivo CSV de db línea por línea y crea una lista de usuarios. Cada línea se convierte en una estructura USUARIO, que se almacena en la memoria compartida el caso de usuario.

3.3.3. 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, en el caso de usuario lo guarda en un puntero que se pasa por referencia y en el de transacción se devuelve la transacción.

3.4. Usuario.c

Gestiona las operaciones principales que puede realizar un usuario dentro del sistema bancario. Las funciones aquí documentadas abarcan desde la inicialización del entorno y la interacción con el menú, hasta operaciones específicas como consultar saldo, realizar transacciones, ingresar o retirar dinero, y cancelar tarjetas.

3.4.1. main

Función principal que inicializa el programa y muestra el menú de opciones.

Parámetros:

- `argc`: número de argumentos de la línea de comandos.
- `argv`: arreglo de argumentos de la línea de comandos.

Flujo del programa:

1. Convierte los argumentos de la línea de comandos a enteros.
2. Mapea la memoria compartida.
3. Muestra el nombre del usuario y espera una tecla para continuar.
4. Inicializa variables globales.
5. Muestra el menú de opciones en un bucle infinito.

3.4.2 CancelarTarjetaHilo

Función de hilo que gestiona la cancelación de una tarjeta bancaria modificando el número de cuenta del usuario.

Parámetros:

- `valor`: puntero genérico que se castea a `IdValor*`, estructura que contiene el ID del usuario y el nuevo número de cuenta.

Flujo del programa:

1. Abre un semáforo nombrado para sincronizar el acceso a recursos compartidos.
2. Espera (bloquea) hasta obtener acceso exclusivo.
3. (Comentado) Se modificaría el número de cuenta del usuario.
4. Libera el semáforo y lo cierra.
5. Libera la memoria dinámica utilizada para los parámetros.
6. Retorna `NULL`.

3.4.3 CancelarTarjeta

Función que solicita al usuario un nuevo número de cuenta y lanza un hilo para procesar la cancelación.

Parámetros:

- `idUser`: puntero al ID del usuario actual.

Flujo del programa:

1. Solicita al usuario un nuevo número de cuenta.
2. Verifica que el formato sea correcto (9 caracteres y numérico).

3. Comprueba que el número de cuenta no esté ya en uso.

4. Prepara los parámetros y lanza el hilo `CancelarTarjetaHilo`.

3.4.4 ConsultarSaldo

Muestra el saldo actual del usuario.

Parámetros:

- `idUser`: puntero al ID del usuario.

Flujo del programa:

1. Verifica que el ID no sea nulo.

2. Limpia la pantalla.

3. Muestra el nombre del usuario y su saldo.

4. Espera a que el usuario presione una tecla para continuar.

3.4.5 InfoCuenta

Muestra información detallada de la cuenta del usuario.

Parámetros:

- `idUser`: puntero al ID del usuario.

Flujo del programa:

1. Verifica que el ID sea válido.

2. Limpia la pantalla.

3. Muestra nombre, contraseña, número de cuenta, saldo e ID.

4. Espera a que el usuario presione una tecla para continuar.

3.4.6 Menú opciones

Función que muestra el menú de opciones disponibles para el usuario y ejecuta la acción correspondiente según la selección. Esta versión utiliza una estructura `IdValor` para manejar tanto el ID del usuario como un valor adicional que puede ser modificado (por ejemplo, para controlar el cierre del programa).

Parámetros:

- `parametros`: puntero a una estructura `IdValor` que contiene:
  - `id`: puntero al ID del usuario.
  - `valor`: puntero a un entero que puede ser modificado (por ejemplo, para indicar si el usuario desea cerrar sesión).

Flujo del programa:

1. Verifica que el puntero `parametros` no sea nulo.
2. Extrae el ID del usuario desde `parametros->id`.
3. Muestra el menú de opciones disponibles.
4. Solicita al usuario que seleccione una opción.
5. Según la opción seleccionada:
   - 1 a 6: Llama a la función correspondiente (`SacarDinero`, `MeterDinero`, `ConsultarSaldo`, `InfoCuenta`, `Transaccion`, `CancelarTarjeta`) y muestra el valor contenido en `parametros->valor`.
   - 7: Resta 1 al valor apuntado por `parametros->valor`, abre un FIFO para notificar el cierre de sesión, escribe el número de cuenta del usuario en el FIFO, edita el CSV del usuario, elimina los semáforos y libera la memoria antes de finalizar el programa con `exit(1)`.
6. Si la opción no es válida, no realiza ninguna acción.

3.4.7 MeterDineroHilo

Función de hilo que incrementa el saldo del usuario y registra la transacción.

Parámetros:

- `valor`: puntero genérico que se castea a `IdValor*`, con ID del usuario y cantidad a ingresar.

Flujo del programa:

1. Verifica que el parámetro no sea nulo.
2. Abre y bloquea el semáforo para acceso exclusivo.
3. Incrementa el saldo del usuario.

4. Crea y registra una transacción.

5. Escribe la transacción en el log.

6. Libera el semáforo y los recursos.

7. Escribe un mensaje en el log general.

8. Retorna `NULL`.

3.4.8 MeterDinero

Solicita una cantidad al usuario y lanza un hilo para ingresarla en su cuenta.

Parámetros:

- `idUser`: puntero al ID del usuario.

Flujo del programa:

1. Solicita al usuario la cantidad a ingresar.

2. Verifica que sea positiva y no supere el límite permitido.

3. Prepara los parámetros y lanza el hilo `MeterDineroHilo`.

3.4.9 RegistrarTransaccion

Registra una transacción en el archivo de transferencias.

Parámetros:

- `transaccion`: puntero a una estructura `TRANSACCION` con los datos de la operación.

Flujo del programa:

1. Obtiene la fecha y hora actual.

2. Verifica que la transacción no sea nula.

3. Abre el archivo de transferencias en modo append.

4. Escribe los datos de la transacción.

5. Cierra el archivo.

3.4.10 _HiloSacarDinero

Función de hilo que gestiona la retirada de dinero de la cuenta de un usuario.

Parámetros:

- `valor`: puntero genérico que se castea a `IdValor*`, estructura que contiene el ID del usuario y la cantidad a retirar.

Flujo del programa:

1. Verifica que el parámetro no sea nulo.
2. Abre y bloquea el semáforo para acceso exclusivo.
3. Comprueba si el usuario tiene saldo suficiente.
4. Si no lo tiene, registra un intento fallido en el log y termina.
5. Si tiene saldo suficiente, lo descuenta del total.
6. Crea y registra una transacción con la descripción "Retirada manual".
7. Escribe la transacción en el log de transacciones.
8. Libera el semáforo y los recursos.
9. Registra la operación en el log general.
10. Retorna `NULL`.

3.4.11 SacarDinero

Función que solicita al usuario una cantidad a retirar y lanza un hilo para procesar la operación.

Parámetros:

- `idUser`: puntero al ID del usuario.

Flujo del programa:

1. Solicita al usuario la cantidad a retirar.
2. Verifica que la cantidad sea válida (no negativa).
3. Prepara los parámetros y lanza el hilo `_HiloSacarDinero`.

3.4.12 TransaccionHilo

Función de hilo que realiza una transferencia entre dos cuentas de usuario.

Parámetros:

- `valor`: puntero genérico que se castea a un arreglo de dos `IdValor`, uno para el usuario emisor y otro para el receptor.

Flujo del programa:

1. Abre y bloquea el semáforo para acceso exclusivo.
2. Verifica que el usuario emisor tenga saldo suficiente.
3. Si no lo tiene, registra un intento fallido en el log y termina.
4. Descuenta la cantidad del emisor y la suma al receptor.
5. Crea y registra una transacción con la descripción "Transferencia entre cuentas".
6. Escribe la transacción en el log de transacciones.
7. Libera el semáforo y los recursos.
8. Registra la operación en el log general.
9. Retorna `NULL`.

3.4.13 Transaccion

Función que solicita los datos para una transferencia y lanza un hilo para ejecutarla.

Parámetros:

- `idUser`: puntero al ID del usuario que realiza la transferencia.

Flujo del programa:

1. Solicita el número de cuenta destino y la cantidad a transferir.
2. Verifica que la cantidad sea válida y no supere el límite permitido.
3. Busca el ID del usuario receptor a partir del número de cuenta.
4. Si no se encuentra, muestra un mensaje de error.
5. Prepara los parámetros y lanza el hilo `TransaccionHilo`.

3.5 Común

Este apartado documenta las funciones comunes utilizadas en el sistema bancario, las cuales proporcionan soporte esencial para la gestión de usuarios, configuración del entorno y registro de eventos. Estas funciones no están directamente ligadas a la interacción del usuario final, pero son fundamentales para el correcto funcionamiento del sistema. Abarcan desde la edición de archivos CSV y la escritura en logs, hasta la inicialización de parámetros globales y la lectura de datos persistentes.

3.5.1 EditarCsv

Función que edita el archivo CSV de usuarios, reemplazando la información de un usuario específico.

Parámetros:

- `usuario`: puntero a una estructura `USER` que contiene la información del usuario a actualizar.

Flujo del programa:

1. Verifica que el puntero `usuario` no sea nulo.
2. Abre el archivo CSV original en modo lectura.
3. Crea un archivo temporal en modo escritura.
4. Lee cada línea del archivo original.
5. Si la línea corresponde al usuario a actualizar, escribe la nueva información en el archivo temporal.
6. Si no, copia la línea original al archivo temporal.
7. Cierra ambos archivos.
8. Reemplaza el archivo original con el archivo temporal.
9. Retorna `NULL`.

3.5.2 EscribirEnLog

Función que escribe un mensaje en el archivo de log, incluyendo la fecha y hora actuales.

Parámetros:

- `frase`: cadena de caracteres que contiene el mensaje a escribir en el log.

Flujo del programa:

1. Obtiene la fecha y hora actuales del sistema.
2. Formatea el mensaje con la fecha y hora.
3. Abre el semáforo para sincronizar el acceso al archivo de log.
4. Espera (bloquea) hasta obtener acceso exclusivo.
5. Abre el archivo de log en modo append.

6. Escribe el mensaje en el archivo de log.

7. Cierra el archivo de log.

8. Libera el semáforo.

3.5.3 InitGlobal

Función que inicializa las variables globales del sistema, leyendo la configuración desde un archivo de propiedades.

Parámetros:

- Ninguno.

Flujo del programa:

1. Inicializa las estadísticas de usuarios a 0.
2. Abre el archivo de propiedades en modo lectura.
3. Elimina cualquier semáforo existente.
4. Lee cada línea del archivo de propiedades.
5. Si la línea contiene una configuración válida, la almacena en la estructura `Config`.
6. Cierra el archivo de propiedades.

3.5.4 LeerCSVNcuenta

Función que busca un usuario en el archivo CSV por su número de cuenta.

Parámetros:

- `ncuenta`: cadena de caracteres que contiene el número de cuenta a buscar.

Flujo del programa:

1. Inicializa un puntero a entero para el índice del usuario.
2. Abre el archivo CSV en modo lectura.
3. Lee cada línea del archivo CSV.
4. Si encuentra un usuario con el número de cuenta especificado, retorna el índice del usuario.
5. Si no lo encuentra, retorna `NULL`.

3.5.5 leerCsv

Función que lee la información de un usuario desde el archivo CSV por su ID de línea.

Parámetros:

- `idLinea`: puntero a un entero que contiene el ID de la línea del usuario a leer.

Flujo del programa:

1. Verifica que el ID de línea sea válido.
2. Abre el archivo CSV en modo lectura.
3. Lee cada línea del archivo CSV hasta llegar a la línea especificada por el ID.
4. Crea una estructura `USER` con la información del usuario.
5. Retorna un puntero a la estructura `USER`.

4. Estructura de ficheros.

Esta vez, aunque la información de los usuarios sigue siendo almacenada en db.csv, las transacciones son almacenadas en estructuras de ficheros alojadas en la carpeta `./transacciones/`. La estructura consiste en varias carpetas, una por usuario (el nombre de la carpeta es el número de cuenta del usuario), con dos ficheros: `transacciones.log` y `config.conf`. El primero sirve para almacenar el log de las transacciones de cada usuario, y el segundo sirve de “marcapáginas” para monitor.

Se puede observar que transacciones.log (el que está fuera de la estructura de ficheros previamente mencionada), sigue en funcionamiento. Esto es porque hemos considerado que tener un archivo con la información centralizada en un sistema de estas dimensiones y volumen de datos puede ser recomendable.

5. Limitaciones & Problemas Encontrados

Las limitaciones encontradas en este proyecto han sido nuestro propio conocimiento del lenguaje de programación C, ya que 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 óptima de realizar el fork() y comunicar ambos procesos.

La deficiente planificación en el diseño del proceso de monitor, al implementar su interfaz de forma separada, derivó en la necesidad de una reestructuración completa del código del módulo de monitor. Esta reestructuración implicó el desarrollo de una interfaz independiente encargada de registrar las anomalías en tiempo real conforme ocurren, en lugar de limitarse únicamente a su registro en el log. Esto permite una detección más proactiva de anomalías, facilita la implementación de mecanismos de respuesta inmediata y mejora la trazabilidad del flujo de datos entre el sistema de monitoreo y otros módulos del sistema operativo.

6. Conclusión

Este programa ha sido un reto sobre todo controlar los distintos ficheros para los distintos usurarios, teniendo que controlar los distintos usuarios y sus interrupciones para que no se escriba información ajena en log de otros usuarios. también ha sido un reto crear la memoria compartida ya no tanto usarla, pero crearla, y dar las llaves a los distintos usuarios ha sido complicado.

Después de esta práctica hemos conseguido aprender bien el uso de la memoria compartida en el programa y la importancia de ajustarla al máximo, liberándola al final de su uso.