Universidad Francisco de Vitoria

Práctica Final de Asignatura – Parte I

Simulador de un Sistema Bancario Concurrente Avanzado SecureBank

1. Enunciado

Eres parte del equipo de desarrollo de **SecureBank**, un innovador banco digital que necesita construir un sistema bancario concurrente para gestionar transacciones en tiempo real. Este sistema debe estar diseñado para aprovechar al máximo las capacidades de Linux como sistema operativo multitarea, y el lenguaje C como herramienta de desarrollo, debido a su capacidad de control de bajo nivel y acceso directo a características del sistema.

El objetivo es construir un sistema bancario modular y robusto que gestione múltiples usuarios y operaciones simultáneas, mientras detecta patrones sospechosos y genera reportes de auditoría en tiempo real. Este sistema utilizará tuberías, paso de mensajes, manejo de señales, y sincronización para garantizar la integridad y seguridad de los datos.

El proyecto se centra en los temas clave de la asignatura, como la gestión de procesos, concurrencia, exclusión mutua, y sincronización, utilizando C y shell scripting para maximizar el control y la eficiencia.

2. Contexto del problema

Desarrollarás un sistema bancario concurrente avanzado que gestione múltiples usuarios realizando transacciones simultáneamente. El sistema debe estar diseñado para:

- Crear procesos hijos para gestionar cada usuario.
- Usar hilos para manejar las operaciones dentro de cada usuario.
- Sincronizar el acceso a los recursos compartidos (archivos de cuentas y logs).
- Detectar y reportar transacciones sospechosas en tiempo real.

El diseño debe utilizar características avanzadas de Linux (*tuberías, señales, semáforos, colas de mensajes*) y el lenguaje C para optimizar el control sobre el sistema.

3. Especificaciones Generales

El sistema consta de los siguientes componentes:

a) *Programa principal (banco.c):*
   - Es el programa central que inicializa el sistema, administra las cuentas y coordina los procesos hijos que representan a los usuarios.
   - Comunica con los procesos hijos mediante tuberías y señales.

b) *Procesos hijos (Usuarios):*
   - Cada usuario es un proceso independiente que interactúa con el sistema mediante un menú interactivo.
   - Dentro de cada proceso hijo, se crean hilos para ejecutar las operaciones bancarias (depósitos, retiros, transferencias).

c) *Sincronización y Exclusión Mutua:*
   - Usar semáforos para sincronizar el acceso al archivo de cuentas.
   - Mutexes para garantizar la exclusión mutua en operaciones concurrentes.

d) *Detección de anomalías:*

- Un proceso independiente (monitor) analiza las transacciones en tiempo real y genera alertas mediante tuberías.

4. Flujo Completo del Sistema

1. Inicio del Sistema

1. **Inicialización del archivo de cuentas:**
   - Un programa auxiliar *init_cuentas.c* crea un archivo binario con datos iniciales de cuentas. Ejemplo de estructura en C:

```
struct Cuenta {
  int numero_cuenta;
  char titular[50];
  float saldo;
  int num_transacciones;
};
```

Formato del archivo:

```
1001,John Doe,5000.00,0
1002,Jane Smith,3000.00,0
```

2. **Inicialización de semáforos:**
   - Se crea un semáforo nombrado usando sem_open para controlar el acceso al archivo de cuentas:

```
sem_t *semaforo = sem_open("/cuentas_sem", O_CREAT, 0644, 1);
```

3. **Lanzamiento del proceso principal:**
   - El programa principal *banco.c* ejecuta un bucle para esperar conexiones de usuarios.
   - Al recibir una solicitud de conexión, genera un proceso hijo:

```
pid_t pid = fork();
if (pid == 0) {
  // Código del proceso hijo (usuario)
  ejecutar_menu_usuario(pipe_padre_hijo);
}
```

2. Gestión de Usuarios (Procesos Hijos)

1. **Menú de Usuario:**
   Cada proceso hijo presenta un menú interactivo:

```
1. Depósito
2. Retiro
3. Transferencia
4. Consultar saldo
5. Salir
```

- El menú es implementado en C con un bucle while que espera la selección del usuario:

```
int opcion;
while (1) {
  printf("1. Depósito\n2. Retiro\n3. Transferencia\n4. Consultar saldo\n5. Salir\n");
  scanf("%d", &opcion);
  switch (opcion) {
    case 1: realizar_deposito(); break;
    case 2: realizar_retiro(); break;
    case 3: realizar_transferencia(); break;
    case 4: consultar_saldo(); break;
    case 5: exit(0);
  }
}
```

2. **Comunicación con el proceso principal:**
   - Cada proceso hijo envía los detalles de la operación al proceso principal mediante una tubería:

```
write(pipe_hijo_padre[1], &operacion, sizeof(operacion));
```

3. Ejecución de Operaciones Bancarias (Hilos)

1. **Hilos por operación:**
   - Dentro del proceso hijo, cada operación se ejecuta como un hilo:

```
pthread_t hilo;
pthread_create(&hilo, NULL, ejecutar_operacion, (void *)&datos_operacion);
pthread_join(hilo, NULL);
```

2. **Protección del archivo de cuentas:**
   - Antes de leer o escribir en el archivo de cuentas, el hilo adquiere el semáforo:

```
sem_wait(semaforo);
actualizar_archivo_cuentas(operacion);
sem_post(semaforo);
```

3. **Actualización de cuentas:**
   - La operación modifica el archivo binario:

```
FILE *archivo = fopen("cuentas.dat", "rb+");
fseek(archivo, posicion_cuenta, SEEK_SET);
fwrite(&cuenta_actualizada, sizeof(Cuenta), 1, archivo);
fclose(archivo);
```

4. Detección de Anomalías

1. **Proceso de monitoreo:**
   - Un proceso independiente (monitor.c) lee las transacciones desde una cola de mensajes:

```
struct msgbuf {
  long tipo;
  char texto[100];
};
msgrcv(cola_mensajes, &mensaje, sizeof(mensaje.texto), 0, 0);
```

2. **Identificación de patrones sospechosos:**
   - El proceso busca:
     - Retiros consecutivos mayores a un monto límite.
     - Transferencias repetitivas entre cuentas.
     - Uso simultáneo de una cuenta desde varios procesos.
3. **Alertas en tiempo real:**
   - Si se detecta una anomalía, el monitor envía una alerta mediante una tubería:

```
write(pipe_monitor_hijos[1], "ALERTA: Transacción sospechosa en cuenta 1002\n", 50);
```

5. Finalización del Sistema

1. **Cierre controlado:**
   - Los semáforos se destruyen al final del programa:

```
sem_unlink("/cuentas_sem");
```

2. **Generación del Log Final:**
   - El proceso principal registra todas las operaciones en un archivo log:

```
[2024-12-01 12:00:00] Depósito en cuenta 1001: +1000.00
[2024-12-01 12:01:00] Retiro en cuenta 1002: -500.00
```

5. Estructura del Sistema

1. **banco.c:**
   - Programa principal que coordina los procesos hijos y el monitor.
2. **usuario.c:**
   - Código del menú interactivo y la gestión de hilos.
3. **monitor.c:**
   - Detección de anomalías y generación de alertas.
4. **init_cuentas.c:**
   - Inicialización del archivo de cuentas.

6. Configuración del sistema mediante ficheros de parametrización

El sistema bancario deberá ser flexible y adaptable. Para ello, utilizaremos un fichero de parametrización que permita modificar valores clave del sistema sin necesidad de recompilar el código. Esto facilitará la configuración inicial y el ajuste de parámetros durante la ejecución.

Especificaciones del fichero de configuración

El fichero de configuración *config.txt* debe incluir los siguientes parámetros:

1. *Límites de operaciones:*
   - LIMITE_RETIRO: Monto máximo permitido por cada retiro.
   - LIMITE_TRANSFERENCIA: Monto máximo permitido por cada transferencia.
2. *Umbrales para detección de anomalías:*
   - UMBRAL_RETIROS: Número de retiros consecutivos altos para generar una alerta.
   - UMBRAL_TRANSFERENCIAS: Número de transferencias repetitivas entre las mismas cuentas para generar una alerta.
3. *Parámetros de ejecución:*
   - NUM_HILOS: Número máximo de hilos simultáneos permitidos por proceso hijo.
   - ARCHIVO_CUENTAS: Ruta al archivo de cuentas bancarias.
   - ARCHIVO_LOG: Ruta al archivo de log.

Ejemplo de config.txt

```
# Límites de Operaciones
LIMITE_RETIRO=5000
LIMITE_TRANSFERENCIA=10000

# Umbrales de Detección de Anomalías
UMBRAL_RETIROS=3
UMBRAL_TRANSFERENCIAS=5

# Parámetros de Ejecución
NUM_HILOS=4
ARCHIVO_CUENTAS=cuentas.dat
ARCHIVO_LOG=transacciones.log
```

Integración en el sistema

1. **Lectura del fichero de configuración**
   - Implementa una función en *banco.c* para leer el fichero config.txt al inicio del programa:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct Config {
    int limite_retiro;
    int limite_transferencia;
    int umbral_retiros;
    int umbral_transferencias;
    int num_hilos;
    char archivo_cuentas[50];
    char archivo_log[50];
} Config;

Config leer_configuracion(const char *ruta) {
    FILE *archivo = fopen(ruta, "r");
    if (archivo == NULL) {
        perror("Error al abrir config.txt");
        exit(1);
    }

    Config config;
    char linea[100];
    while (fgets(linea, sizeof(linea), archivo)) {
        if (linea[0] == '#' || strlen(linea) < 3) continue; // Ignorar comentarios y
líneas vacías
        if (strstr(linea, "LIMITE_RETIRO")) sscanf(linea, "LIMITE_RETIRO=%d",
&config.limite_retiro);
        else if (strstr(linea, "LIMITE_TRANSFERENCIA")) sscanf(linea,
"LIMITE_TRANSFERENCIA=%d", &config.limite_transferencia);
        else if (strstr(linea, "UMBRAL_RETIROS")) sscanf(linea, "UMBRAL_RETIROS=%d",
&config.umbral_retiros);
        else if (strstr(linea, "UMBRAL_TRANSFERENCIAS")) sscanf(linea,
"UMBRAL_TRANSFERENCIAS=%d", &config.umbral_transferencias);
        else if (strstr(linea, "NUM_HILOS")) sscanf(linea, "NUM_HILOS=%d",
&config.num_hilos);
        else if (strstr(linea, "ARCHIVO_CUENTAS")) sscanf(linea, "ARCHIVO_CUENTAS=%s",
config.archivo_cuentas);
        else if (strstr(linea, "ARCHIVO_LOG")) sscanf(linea, "ARCHIVO_LOG=%s",
config.archivo_log);
    }

    fclose(archivo);
    return config;
}
```

2. **Inicialización del sistema con parámetros**
    - Llama a *leer_configuracion* en el programa principal:

```
Config configuracion = leer_configuracion("config.txt");
printf("Archivo de cuentas: %s\n", configuracion.archivo_cuentas);
printf("Número máximo de hilos: %d\n", configuracion.num_hilos);
```

3. **Uso de los parámetros en el sistema**
    - Limitar retiros y transferencias:

```
if (monto > configuracion.limite_retiro) {
   printf("Error: Retiro excede el límite permitido (%d)\n", configuracion.limite_retiro);
   return;
}
```

- Configurar el número máximo de hilos:

```
pthread_t hilos[configuracion.num_hilos];
```

- Especificar rutas dinámicamente:

```
FILE *archivo = fopen(configuracion.archivo_cuentas, "rb+");
```

4. **Notificación de anomalías basadas en umbrales**
   - Garantiza que el proceso de detección de anomalías *monitor.c* use los umbrales configurados:

```
if (retiros_consecutivos > configuracion.umbral_retiros) {
   enviar_alerta("ALERTA: Retiros consecutivos detectados.");
}
```

Entrega Final

1. Código modularizado y documentado.
   - **(30% de la nota)** El código fuente de la solución funcionando.
2. Archivo log con transacciones y anomalías.
   - **(10% de la nota)** Junto con el plan de la aplicación.
3. Manual técnico que explique el diseño del sistema.
   - **(40% de la nota)** Memoria de la práctica en PDF. La memoria debe incluir:
     - Elementos comunes de todo trabajo (portada, índice, apartados organizados, etc.)
     - Explicación de la solución.
     - Esquema generar de la aplicación.
     - Capturas de su funcionamiento.
4. Presentación y defensa de la práctica
   - **(20% de la nota)**

Evaluación mediante defensa

- El grupo debe preparar la presentación de su solución, ejecutar el código e ir explicando el plan de pruebas que haya realizado.
- El profesor durante la exposición revisará la memoria e irá haciendo preguntas sobre la solución desarrollada.
- El profesor podrá pedir modificar la práctica (cambiar algo en la configuración, por ejemplo) y el equipo tendrá que demostrar que conoce su práctica y sabe como mostrar su ejecución real.
- En el caso de que la defensa se haga en público, los grupos presentes en la presentación podrán preguntar, aportar críticas o sugerencias sobre la solución.
- Es obligatorio aprobar la revisión presencial y defender correctamente la práctica.

Universidad Francisco de Vitoria

Práctica Final de Asignatura – Parte I

Simulador de un Sistema Bancario Concurrente Avanzado SecureBank

Consideraciones importantes

a) *El código debe compilar sin errores y funcionar con la configuración que indique el fichero.*
b) *No se aceptan mensajes de "warning" al compilar.*
c) *El código debe ir correctamente documentado y/o comentado, explicando qué hace cada función desarrollada.*
d) *El código debe estar optimizado, sin redundancia de código, bucles innecesarios o condiciones mal definidos (if repetidos por todo el código).*
e) *Se valora el uso de una interfaz que le indique al usuario de qué trata la aplicación.*
f) *En la memoria, utiliza capturas de pantalla donde se muestre el funcionamiento del sistema.*