Tarea 1

Implementación de un sistema transaccional

Publicada el 2 de septiembre de 2017

Fecha de entrega (parte 2): viernes 13 de septiembre de 2017, 23:59 hrs., en Git.

1. Objetivo

El objetivo de la segunda parte de la tarea es agregar soporte para simulación de múltiples terminales en cada sucursal utilizando *threads*, y proveer sincronización tanto para las operaciones en las cuentas bancarias de clientes, como para las operaciones de volcado (*dump*) de las cuentas y transacciones.

2. Prerequisitos

Es importante completar la implementación de procesos de sucursal que fue requerida en la primera parte. Para comenzar la implementación de la segunda parte, cada sucursal debe contener un thread que genere transacciones aleatorias y otro thread encargado de la comunicación con la oficina matriz. Además, se debe contar con la implementación de la comunicación entre la oficina matriz y las sucursales por mensajes vía *pipes*.

Además del dominio de las APIs POSIX de procesos utilizadas en la parte anterior, es muy importante tener familiaridad con la API de *threads* POSIX (`pthread`). Esta API está descrita en los capítulos 26 a 31 del libro Arpaci-Dusseau.

3. Descripción y Requisitos

A continuación se presentan definiciones actualizadas de acuerdo a lo requerido en la segunda parte de la tarea.

- Proceso principal
  - En *thread* principal: Interfaz de consola de texto para ingreso de comandos.
  - En *threads* concurrentes: Uno o más threads que ejecutan funciones que simulan la oficina matriz del banco, incluyendo la comunicación con sucursales.
- Proceso de sucursal

- En *thread* principal: Mantiene la comunicación con la oficina matriz a través de mensajería por pipes.
- En *threads* concurrentes: Se deben implementar threads que simulen terminales bancarios desde donde se realizan operaciones de depósito y retiro de dinero hacia/desde cuentas inscritas en el banco. Además, es recomendable que las operaciones de *dump* sean procesadas por un thread concurrente exclusivo.

Al igual que en la primera parte de la tarea, en el sistema hay un único proceso principal y uno o más procesos de sucursal.

3.1. Nueva Funcionalidad

Dado que en la segunda parte de la tarea una sucursal puede contener muchos terminales concurrentes generando transacciones aleatorias, es fundamental proveer acceso sincronizado a las estructuras de datos que los *threads* comparten. En particular, cada sucursal debe proveer exclusión mutua para acceso a las cuentas individuales, y exclusión mutua para acceder al arreglo o estructura de datos que contiene todas las cuentas. Para lograr lo primero, deben crearse un *mutex* para cada cuenta bancaria. Para lo segundo, bastará un *mutex* para garantizar exclusión mutua cuando se accede a la estructura de datos compartida y se requiere, por ejemplo, ejecutar una operación de *dump*.

Aparte de la necesidad de implementar sincronización para el funcionamiento de *threads* en sucursales, a continuación se describe nueva funcionalidad que debe implementarse al ejecutar los comandos que se introducen en la interfaz de línea de comandos en el proceso de la oficina matriz:

`quit`: Termina el proceso de oficina matriz y los procesos de sucursales. El proceso matriz debe esperar que los procesos de sucursales terminen, usando la llamada al sistema `wait`. Los procesos de sucursales deben esperar la terminación de los *threads* que contengan.

`init`: Inicia una sucursal. Además del valor N que determina el número de cuentas que la sucursal debe mantener, se debe agregar el valor entero T, $1 \leq T \leq 8$, el cual dice cuántos terminales bancarios deben crearse en la sucursal.

`kill`: Detiene una sucursal. Permite terminar una sucursal especificando su ID. La sucursal debe terminar limpiamente, es decir, esperando la terminación de todos los *threads* que ejecutan en ella.

`list`: Imprime una tabla con las sucursales creadas, indicando el ID de cada una, junto con el rango de números de cuenta válidos en cada una, y el número de terminales contenidos en cada sucursal.

`dump`: Se invoca especificando el ID de una sucursal. La sucursal indicada debe generar un respaldo de todas las transacciones que se han originado en ella en formato CSV[1], con nombre de archivo

[1] Archivo de valores separados por coma (*Comma Separated Values*)

`dump_PID.csv`. Columnas: tipo de transacción, medio de origen, cuenta de origen, cuenta de destino.

`dump_accs`: Se invoca especificando el ID de una sucursal. La sucursal indicada debe generar un respaldo del saldo de todas las cuentas bancarias inscritas en ella, en formato CSV, con nombre de archivo `dump_accs_PID.csv`. Columnas: número de cuenta, saldo.

`dump_errs`: Se invoca especificando el ID de una sucursal. La sucursal indicada debe generar un respaldo de todas las transacciones que han generado error, por falta de saldo o por número de cuenta no válido. El nombre de archivo debe ser `dump_errs_PID.csv`. Columnas: tipo de error (numerar con 1 y 2 los dos casos ya descritos), número de cuenta, saldo previo a la transacción, monto que se quiso retirar. En caso que el error sea por número de cuenta no válido, los dos últimos campos pueden mantenerse vacío.

3.2. Consideraciones Técnicas

Para desarrollar esta tarea necesitarás usar funciones de la librería estándar de C, y también funciones de la interfaz de programación de llamadas de sistema[2]. Algunas de las llamadas de sistema y funciones relevantes de la biblioteca estándar de C a utilizar podrían ser las siguientes (revisar la documentación de cada una):

- `fopen`, `fgets`, `getline`, `feof`, `fprintf`, `fclose`: manejo de archivos.
- `fork`, `wait`, `getpid`, `pipe`: creación y administración de procesos y *pipes*.
- `malloc`, `realloc`, `free`: manejo de memoria dinámica.
- `strnlen`, `strncmp`, `strncpy`, `snprintf`: manejo de *strings*.
- `perror`, `strerror`: manejo de errores.

Se recomienda usar comandos `ps` y `kill` para listar y matar manualmente procesos de sucursales creados con `init` mientras se implementan las funciones de la tarea.

El comando *dump* requiere guardar el historial de transacciones generadas en cada sucursal. Para esto, se puede utilizar una estructura (`struct`) para representar las transacciones y una lista ligada para mantenerlas disponibles en memoria.

Cuando una sucursal es iniciada desde el proceso principal, el proceso principal puede (y debe) conocer los números de cuentas válidos en las sucursales. En consecuencia, el proceso principal puede validar si una transacción es posible dado su número de cuenta de destino. Si una transacción falla porque la cuenta de destino no tiene los fondos suficientes, entonces la sucursal debe informar a la oficina principal para que quede registro del error.

[2] *POSIX system calls API*, hay una buena lista en `http://www.tutorialspoint.com/unix_system_calls/index.htm`

Las cuentas bancarias pueden ser implementadas en cada sucursal mediante un arreglo de enteros positivos (las cuentas no tienen saldo negativo).

4. Evaluación

4.1. Inspección de código y funcionamiento (50 %)

Se revisará el código de la tarea en forma exhaustiva y se realizarán pruebas de funcionamiento con cada uno de los comandos contemplados. Se revisarán los datos generados en las operaciones de *dump*.

4.2. Manejo de errores (10 %)

Un proceso de sucursal que se cae no debe comprometer el funcionamiento del proceso principal del sistema. Además, se deben manejar adecuadamente errores en los siguientes casos:

- Comando no válido ingresado por el usuario en la consola.
- Error al abrir un archivo para escritura.
- Error al crear un pipe.
- Invocación a `malloc` retorna `NULL`.
- Invocación a `fork` retorna `-1`.
- Fallan funciones del API de POSIX *threads*.

4.3. Manejo de memoria (15 %)

Es importante que toda la memoria pedida por el programa al sistema operativo a través de invocaciones a funciones como `malloc` y `realloc` sea correctamente liberada llamando a `free`, y que además, todo el acceso a la memoria a través de punteros sea implementado en forma correcta.

4.4. Informe (25 %)

La segunda parte de la tarea debe incluir un informe conciso en un archivo de texto de nombre `README`, ubicado en el directorio raíz del código de la tarea. El archivo debe contener secciones separadas para documentar los siguientes aspectos:

- Descripción de las funciones principales del programa, tanto del proceso de casa matriz, como las sucursales.

- Problemas encontrados en la implementación del sistema, y en particular, problemas de concurrencia y sincronización que se hayan suscitado con el uso de *threads*. En particular, si el programa se cae en algún punto de la ejecución, describir dónde está la mayor sospecha del problema que causa la caída.
- Funciones del sistema sin implementar. Describir aquellas funciones que no pudieron implementarse por razones de complejidad y/o tiempo.

4.5. Bonus opcional (1 punto adicional en la nota)

Pueden realizar un estudio que analice la cantidad de transacciones totales generadas en función de la cantidad de terminales en las sucursales. Idealmente, ilustrar cómo la oficina matriz puede llegar a actuar como cuello de botella para el procesamiento de las transacciones cuando se aumenta el número de terminales y la frecuencia de las transacciones.

Este estudio pueden entregarlo con posterioridad a la fecha de entrega de la segunda parte, con fecha de entrega flexible, para mejorar su nota final en la tarea 1 hasta en un punto (nota máxima 8,0).

4.6. Modalidad de Trabajo

Se mantendrán los grupos de la parte 1 y podrán continuar trabajando en el mismo repositorio GitHub utilizado hasta el momento. Si hubiera alumnos que deseen cambiar de compañero de equipo, pueden hablarlo directamente con el profesor.

5. Entrega

La tarea debe compilar; en caso contrario, la nota máxima será un 3,9 – apelable – si el error de compilación es un detalle mínimo. La entrega debe hacerse en un archivo zip que debe contener el programa `main.c` y un archivo `Makefile` que permita compilarlo con `make`.

El lenguaje C tiene una serie de dialectos y extensiones según la versión de las herramientas que se esté usando para desarrollar. Para evitar confusiones, la evaluación será hecha compilando con `gcc` (la versión incluída en la distribución de Ubuntu de referencia), usando los parámetros `-std=c11 -Wall -Werror`.

La entrega de la tarea debe hacerse hasta las 23:59 hrs. en el repositorio Git del grupo. Para hacer efectiva esta restricción, se evaluará la última revisión del repositorio del grupo previa a la hora de cierre. Si un grupo hace su primer *commit* y/o *push* después de la hora de cierre, tendrá derecho a nota máxima 4,0.