Tarea 1

Implementación de un sistema transaccional

Publicada el 17 de agosto de 2017

Fecha de entrega (parte 1): viernes 31 de agosto de 2017, 23:59 hrs., en Git.

1. Objetivo

El objetivo de esta tarea implementar de un sistema transaccional que simule el funcionamiento de un banco, para aplicar los contenidos de procesos y *threads* vistos en el curso.

La tarea tiene dos partes. En la primera parte se pide implementar el funcionamiento general de los procesos involucrados en el sistema, y en la segunda, el énfasis estará en implementar *threads* en los procesos para paralelizar operaciones bancarias. El enunciado de la segunda parte estará publicado días antes de la entrega de la primera parte.

2. Prerequisitos

Para esta tarea es importante repasar programación en C, incluyendo entrada y salida en consola y archivos, control de flujo, arreglos, strings, punteros y manejo de memoria. Para esto, puedes repasar los ejercicios del libro "*Learning C Programming the Hard Way*", que hemos visto en clases y que está disponible en SAF. También el libro "*Essential C*" es una buena referencia para comprender los detalles del lenguaje C. Finalmente, el capítulo 14 del libro de Arpaci-Dusseau describe el funcionamiento de la API de memoria en C.

Además, es indispensable contar con un ambiente de programación que incluya las herramientas de desarrollo GNU, como `gcc` (compilador de C), `valgrind` (análisis de memoria), `gdb` (depurador), `make` (scripts para compilación), y también `Git` para versionar adecuadamente tu código. El sistema operativo oficial para las tareas del curso será Ubuntu 16.04 LTS `http://www.ubuntu.com/download/desktop`. Puedes instalar Ubuntu rápidamente en tu computador como una máquina virtual con productos como Virtualbox, VMWare o Parallels. Sin embargo, es factible desarrollar la tarea en otros ambientes, como en Cygwin y en Mac OS X, mientras el código finalmente compile correctamente en Ubuntu 16.04.

3. Descripción y Requisitos

3.1. Procesos

El sistema tendrá dos tipos de procesos:

- Proceso principal
  - En *thread* principal: Interfaz de consola de texto para ingreso de comandos.
  - En *thread* concurrente: Ejecución de funciones que simulan la oficina matriz del banco.
- Proceso de sucursal
  - En *thread* principal: Simula un terminal bancario desde donde se realizan operaciones de depósito y retiro de dinero desde cuentas inscritas en el banco.
  - En *thread* concurrente: Simula un terminal bancario desde donde se realizan operaciones de depósito y retiro de dinero desde cuentas inscritas en el banco.

En el sistema hay un único proceso principal y uno o más procesos de sucursal. Los procesos de sucursal son iniciados y detenidos por el usuario desde la interfaz de consola de texto en el proceso principal, como se describirá más adelante.

En la primera parte de la tarea, las sucursales tendrán un único *thread* que simulará terminal y realizará respaldo de las transacciones realizadas. En la segunda parte de la tarea, una sucursal podrá simular múltiples terminales, con *threads* concurrentes que habrá que sincronizar debidamente.

3.2. Comunicación entre procesos

Los procesos de sucursales no pueden comunicarse directamente en la eventualidad de que una operación bancaria requiera revisar o modificar el saldo en una cuenta inscrita en otra sucursal. Las transacciones entre sucursales distintas deben ser intermediadas por el proceso en la oficina matriz. En consecuencia, si una transacción bancaria requiere que el depósito o retiro realizado se concrete en una sucursal distinta, la oficina matriz debe operar como nexo entre las dos sucursales.

La oficina matriz mantendrá comunicación con cada sucursal a través de *pipes*. Dado que los *pipes* son unidireccionales, la comunicación bidireccional entre cada sucursal y la oficina matriz requerirá un par de *pipes* exclusivos, de sucursal a matriz y de matriz a sucursal.

3.3. Requisitos de la simulación

Las cuentas bancarias en el sistema tienen un formato de dígitos con tres partes:

```
BBBBBB-SSS-CCCCCC
```

La primera parte `BBB` corresponde al código del banco (los tres últimos dígitos del *Process ID* PID del proceso principal). El sistema simula un solo banco, por lo que el número del banco es determinado por el programa principal al iniciarse. La parte `SSS` corresponde al identificador de una sucursal (al igual que el identificador del banco los tres últimos dígitos del *Process ID* PID del proceso de la sucursal). Finalmente, la parte `CCCCCC` corresponde al número de una cuenta válidamente

registrada en una sucursal. Las partes del identificador ocupan todos los dígitos definidos, rellenando cada parte con ceros por la izquierda de ser necesario[1].

Cuando las una sucursal es iniciada, se crean las cuentas con saldos aleatorios entre $1.000 y $500.000.000 (quinientos millones). Las cuentas mantienen saldo mayor o igual a cero. Si no hay suficiente saldo para realizar una operación, la operación debe ser cancelada.

La sucursal recién creada comienza a generar transacciones con tiempo variable (*random*) entre ellas de 100 a 500 milisegundos.

Las operaciones bancarias posibles son las siguientes:

- Retiro de dinero: Requiere cuenta de origen de los fondos y el monto a retirar.
- Depósito de dinero: Requiere medio de origen (cuenta desde donde se retira o efectivo), y monto a depositar.

3.4. Implementación de la interfaz de usuario y operaciones requeridas

La interfaz de usuario debe mostrar un símbolo al usuario (un *prompt*), como por ejemplo , o `$$` en donde el usuario pueda ingresar comandos. Los comandos permitidos en la interfaz son los siguientes:

`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`.

`init`: Inicia una sucursal. Permite especificar el número de cuentas N (1000 por defecto si no se especifica). Al crear la sucursal imprime el ID de ella (últimos tres dígitos del *Process ID* PID del proceso de la sucursal). Las cuentas en la sucursal se pueden numerar de 1 a N. Se debe usar la llamada al sistema `fork`.

`kill`: Detiene una sucursal. Permite terminar una sucursal especificando su ID.

`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.

`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[2], con nombre de archivo `dump_PID.csv`. Columnas: tipo de transacción, medio de origen, cuenta de origen, cuenta de destino.

[1] Se recomienda usar función `sprintf` para generarlos.
[2] Archivo de valores separados por coma (*Comma Separated Values*)

`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.

3.5. 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[3]. 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.

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

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

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.
- Invocación a `malloc` retorna `NULL`.
- Invocación a `fork` retorna `-1`.

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 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. 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. Modalidad de Trabajo

La tarea debe ser desarrollada en grupos de 2 (parejas). Cada integrante debe contar con una cuenta de usuario en GitHub. El desarrollo debe realizarse por ambos integrantes, y deben quedar claramente registradas sus operaciones de *commit* en el sistema, con comentarios descriptivos en cada una de estas operaciones.

El profesor publicará en Canvas el medio a través del cual se deben inscribir los grupos e informar las cuentas de usuario en GitHub.

Se proporcionará junto con este enunciado un código base con algún trabajo adelantado. Para compilar la tarea basta invocar `make` en el directorio en donde se encuentra el código. El nombre del programa ejecutable es `bank`.

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.