EJERCICIO EVALUABLE 1

Álvaro Obies - 100472136
Carlos Seguí - 100472060

Diseño:

- **Lado del cliente:**
  - client.c:
  - claves.c
    - init
    - set_value
    - get_value
    - modify_value
    - delete_key
    - exist
    - close_server
- **Lado del servidor - server.c:**
  - main
  - init
  - petition_handler
  - exists
  - set_value
  - get_value
  - modify_value
  - delete_key
- **Código en común - utils.h**
  - tuple
  - petition
  - answer

Compilación

Detalles y conclusión

Diseño:

Lado del cliente:

client.c:

El programa *client* tiene la función de permitir acceder a la interfaz de *libclaves.so*. Por lo tanto, su funcionalidad es la más simple de todos los programas de este ejercicio.

En primer lugar, inicia el servicio de la librería empleando su función *init*, y le pide una operación al usuario (en un bucle, de forma que si la operación no es aceptada por *correct_operation* se le pedirá continuamente). Entonces, llamará a la funcion *handle* de la operacion correspondiente. Si esta funcion devuelve -1, llamara a *exit_with_error*, que imprimira un mensaje de error y llamara a *close_server*.
Las funciones *handle* de cada operacion llaman a las funciones de la biblioteca correspondientes, imprimiendo el valor de *get*, pidiendo al usuario el valor de *set*, o el estado de la tupla en el caso de *exist*.

claves.c

Este programa es el que tiene la mayoría de la funcionalidad del lado del cliente. Tiene varias funciones que son llamadas para realizar las operaciones con colas de mensajes.

init

Esta función inicia las colas de mensajes, abriendo la cola del servidor y creando la cola del usuario con un nombre único (cortesía de *getpid*). La cola del servidor es creada para enviar mensajes de tipo *petition*, y la del cliente para mensajes de tipo *answer* (ambos serán vistos más adelantes, cuando se hable de *utils.h*).

set_value

Esta función le pide al usuario todos los valores necesarios para generar un tuple (la clave, el *char* value_1 (asegurándose de que no contenga ninguna coma, ya que el servidor separa estos valores con comas, y los separaría incorrectamente en el caso de que este tuviese alguna), el tamaño del vector (*N_value2*) y los números contenidos en el vector *V_value2*.
Entonces, crea una estructura de tipo *petition*, donde pone la operación "set" y la tupla recién creada, y la envía al servidor.
Una vez hecho esto, espera la respuesta del servidor en formato *answer*. Cuando la ha recibido, comprueba si el *error_code* de esta respuesta es "error" o "noerror", devolviendo -2 o 0 en cada caso (-1 lo devuelve en caso de que haya algún error con las colas).

get_value

Esta función le pide al usuario la clave de la tupla que quiere solicitar. Entonces, crea una *petition* con la operación *get* y una tupla con la clave correspondiente y el resto de valores en blanco.
Después de esto, espera la respuesta del servidor (almacenándola en una *answer*) y comprueba su *error_code*. Si este es "noerror", imprime por pantalla los valores de la *return_tuple* de la respuesta del servidor.

modify_value

Esta función hace lo mismo que set_value, con la diferencia de que en la *petition*, lo que pone en *operation* es "modify".

delete_key

Esta función le pide la clave al usuario, y la almacena en una tupla que almacena en una *petition*, junto con la *operation* "delete". Entonces comprueba la respuesta del servidor (*error_code* en *answer*).

exist

Esta función crea una *petition operation* = "exist". Entonces, la envía al servidor y comprueba si el *error_code* de su respuesta es "exist", "noexist" o "error", imprimiendo lo que corresponda en pantalla.

close_server

Esta función es usada para cerrar el lado del cliente y el del servidor síncronamente.
Lo que hace esta función es enviar una *petition* con la operación "exit", para que este también se cierre. Entonces, cierra las colas del servidor y cliente, y elimina la de cliente. Una vez hecho esto, finaliza la ejecución del programa.

Lado del servidor - server.c:

main

En la función principal, se inicializa la cola del servidor, las propiedades del thread que se encargará de atender las peticiones del cliente y los mutex que se encargar de protegerlas, además de llamar a *init*. Una vez hecho esto, se recibe la petición del cliente actual, se bloquea el mutex y se crea una thread con la función *petition_handler*. Entonces, se espera que el thread use el signal del mutex y se desbloquea. Entonces, se imprime el resultado del anterior thread. Si *petition_handler* devuelve 2, quiere decir que el cliente desea finalizar la conexión, por lo que se cierra la cola del cliente, se destruyen los mutex y se terminal la ejecución.

init

Con esta función abre para escritura el archivo de almacén de tuplas. De esta forma, si no estaba se vuelve a crear, y si estaba se vacía el contenido.

petition_handler

Esta función se encarga de atender las peticiones del cliente. Primero, copian la petición en una variable local (lanzando la señal mutex correspondiente). Luego, mira que operación tiene esta petición, llamando a la función correspondiente y atendiendo los valores que responden estas funciones. Entonces, devuelven la estructura *answer* al cliente.

exists

Esta función abre el archivo de tuplas, y lo lee línea por línea. En cada línea, lee el valor de la clave y lo compara con el valor de la clave solicitado por el cliente. Si es el mismo, devuelve 1; si llega al final sin que esta comparación se realice, devuelve 0.

set_value

Esta función abre el archivo de tuplas en modo *append*, y escribe la tupla solicitada por el usuario (de forma que se escribirá al final del archivo).

get_value

Al igual que *exists*, esta función lee el archivo de tuplas línea por línea, comparando el valor de la clave con el de la clave solicitada. Pero a diferencia de *exists*, esta función lee todos los valores de la línea, y cuando llega la clave solicitada, devuelve estos valores.

modify_value

Esta función abre el archivo de tuplas, y un archivo temporal que abre para escritura, creándolo si no existe (que es lo más probable). Entonces, va leyendo el archivo de tuplas línea por línea. Si la clave de la línea es distinta a la solicitada, escribe el valor anterior de la línea en el archivo temporal. Si es igual a la solicitada, escribe el nuevo valor dado por el cliente. Al final, elimina el archivo anterior y renombra el temporal al mismo nombre del archivo de tuplas, efectivamente convirtiéndolo en el nuevo archivo de tuplas.

delete_key

Esta función es muy similar a *modify_value*. La única diferencia es que cuando la clave leída del archivo de tuplas es la misma a la dada por el cliente, lo que hace esta función no es escribir un valor modificado, simplemente se salta la línea.

Código en común - utils.h

Este archivo simplemente pone a disposición, tanto del cliente como del servidor, de tres estructuras de datos: *tuple, petition* y *answer*.

tuple

key: la clave identificadora única de cada tupla
value1: un *string* (con la restricción de que no puede llevar comas, y tiene el tamaño máximo de 256 bytes).
N_value2: un entero que indica el tamaño del array *V_value2* (con la restricción de que tiene que estar entre 1 y 32).
V_value2: el vector de *floats* de tamaño *N_value2*.

petition

operation: un *string* indicando la operación a realizar
operated_tuple: una *tuple* sobre la que hacer la operación en caso necesario
queue_name: un *string* con el nombre de la cola del cliente

answer

error_code: Un *string* donde poner cosas como "error", "noerror", "exists"...
return_tuple: La tupla a devolver al cliente, para el caso de *get*,

Compilación

Hay dos formas de compilar:
Una es la por defecto, usando el archivo de librería compartida *libclaves.so*:
`make`

Otra, útil en caso de que la librería compartida de error, es usando directamente *claves.c* y *claves.h*:
`make nolib`

Detalles y conclusión

Todas las funciones que necesitan usar el archivo de tuplas en el lado del servidor (exist, set, get, modify, delete…) emplean el mutex file_lock, bloqueándolo antes de abrir el archivo y desbloqueándolo después de cerrarlo, para que el archivo no se convierta en una condición de carrera.
Se decidió usar estructuras fijas para la petición/respuesta a pesar de que en muchas ocasiones se empleaba más espacio del necesario porque daba problemas enviar mensajes con tamaño menor a *mq_msgsize*.