Computer Architecture and Technology Area

University Carlos III of Madrid

OPERATING SYSTEMS

Assignment 2. Concurrency

Degree in Computer Science and Engineering

Index

1. Introduction

The objective of this assignment is to provide a practical view of the synchronization mechanisms available in the POSIX threads library. In particular, this assignment focuses on the basic structures of mutexes and conditional variables as a mean to control the access to critical sections of the program. In this respect, the use of mutexes permits to enforce mutual exclusion when several threads try to access the critical region at the same time. Condition variables on the other hand permit to wait until a certain condition is met.

The synchronization will be implemented using POSIX threads syscalls. In particular:

- ***pthread_create*** will be used to create and execute new threads.
- ***pthread_join*** will be used if any thread must wait for the finalization of another thread.
- Mutexes (***pthread_mutex_init***, ***pthread_mutex_lock***, ***pthread_mutex_unlock,*** and ***pthread_mutex_destroy***) will be used to protect the access to critical sections.
- Cond. variables (***pthread_cond_init***, ***pthread_cond_wait***, ***pthread_cond_signal,*** and ***pthread_cond_destroy***) will be used to wait until a certain condition is met.

The student must design and implement a concurrent access library operating on the top of a warehouse database, in order to guarantee the consistence of all operations even if those operations are performed in a concurrent manner.

.

1.1. Description

The starting point is a "pseudo" database named **db_warehouse**. This database is a C library that allows to manage products in a warehouse (create products, and modify the stock of a concrete product).

On the top of this warehouse, acting as an interface between *db_warehouse* and client threads, there is a library called **sequential**. This library receives request from client threads, and forwards this request to the warehouse. The problem is that this sequential library is not thread-safe. There is a risk of corrupting the database if several threads perform operations at the same time.

The student must implement the library called **concurrent**. This library must be a copy of sequential, but concurrent accesses must be controlled in order not to corrupt the database. The access restrictions that the library must fulfill are the following:

- There are two types of operation: READ (get stock, get number of products) and WRITE (update stock, create new product, etc.).
- An operation may have two different scopes: GLOBAL (the whole database) or LOCAL (to a product).
- GLOBAL WRITE does not allow any other operation over the database.
- GLOBAL READ over the database can be performed concurrently up to MAX_READERS (*concurrent.h*).
- LOCAL WRITE does not allow any other operation over that concrete product, but allows local operations over other products.
- LOCAL READ over that concrete product can be performed concurrently. Besides, local operations over other products are also allowed.

1.1.1. Database interface

The interface to access **db_warehouse** is detailed in this section. The provided library is already compiled, so students cannot access to the source code (but they can access to the header file).

- `int db_warehouse_init()`
    - **Description:** initializes the database. Use once at the beginning.
    - **Return value:** 0 → ok, -1 → error

- `int db_warehouse_destroy()`
    - **Description:** frees the resources of the database. Use once at the ending.
    - **Return value:** 0 → ok, -1 → error

- `int db_warehouse_create_product (char *product_name)`
    - **Description:** stores a new product in the warehouse. It doesn't check if that product already exists.
    - **Input:** product name.
    - **Return value:** 0 → ok, -1 → error there is no space. (16 products max).

- `int db_warehouse_get_num_products(int *num_products)`
    - **Description:** gets the number of products in the warehouse.
    - **Output:** number of products.
    - **Return value:** 0 → ok, -1 → error

- `int db_warehouse_delete_product(char *product_name)`
    - **Description:** deletes from the warehouse the first matching product (the whole product including all its stock).
    - **Input:** product name.
    - **Return value:** 0 → ok, -1 → error product not found.

- `int db_warehouse_exists_product(char *product_name)`
    - **Description:** checks if a product exists.

  - **Input:** product name to be checked.
  - **Return value:** 1 → exists, 0 → doesn't exist

- `int db_warehouse_update_stock(char *product_name, int stock)`
  - **Description:** updates the stock of a product.
  - **Input:** product name and stock.
  - **Return value:** 0 → ok, -1 → error product not found

- `int db_warehouse_get_stock(char *product_name, int *stock)`
  - **Description:** gets the stock of a product.
  - **Input:** product name.
  - **Output:** product stock.
  - **Return value:** 0 → ok, -1 → error product not found.

- `int db_warehouse_set_internal_data(char *product_name, void *ptr, int size)`
  - **Description:** allows to associate to a product internal data defined by the user. It can be used to store structures necessary to synchronize concurrent accesses to a product.
  - **Input:** product name, pointer to the data and size of the data.
  - **Return value:** 0 → ok, -1 → error product not found.

- `int db_warehouse_get_internal_data(char *product_name, void **ptr, int *size)`
  - **Description:** returns the internal data associated to a product.
  - **Input:** product name.
  - **Output:** pointer to the data and size of the data.
  - **Return value:** 0 → ok, -1 → error product not found.

1.1.2. Sequential library

Sequential library is already implemented. It is provided with the aim of showing an example of how a library can act as a wrapper of the database interface. This library already includes all features requested apart from the synchronization mechanism required to guarantee consistency. The student must use this code as example when implementing the concurrent library.

- `int sequential_init()`
    - **Description:** initializes the database and the library resources. Use once at the beginning.
    - **Return value:** 0 → ok, -1 → error

- `int sequential_destroy()`
    - **Description:** frees the resources of the database and the resources used by the library. Use once at the end of the program.
    - **Return value:** 0→ ok, -1→ error

- `int sequential_create_product(char *product_name)`
    - **Description:** stores a new product in the warehouse. If the product already exists the operation is considered successful.
    - **Input:** product name.
    - **Return value:** 0 → ok, -1→ error

- `int sequential_get_num_products(int *num_products)`
    - **Description:** gets the number of products in the warehouse.
    - **Output:** number of products.
    - **Return value:** 0→ ok, -1 → error

- `int sequential_delete_product(char *product_name)`
    - **Description:** deletes from the warehouse the first matching product (the whole product including all its stock). If the product is not found the operation is considered successful.
    - **Input:** product name.

  - **Return value:** 0 → ok, -1→ error

- `int sequential_increment_stock(char *product_name, int stock, int *updated_stock)`
  - **Description:** increments the stock of a product and returns the updated stock.
  - **Input:** product name and amount of stock to be added.
  - **Output:** updated stock.
  - **Return value:** 0 → ok, -1 → error (product not found, etc.)

- `int sequential_decrement_stock(char *product_name, int stock, int *updated_stock)`
  - **Description** decrements the stock of a product and returns the updated stock.
  - **Input:** product name and amount of stock to be subtracted.
  - **Output:** updated stock.
  - **Return value:** 0→ ok, -1 → error (product not found, etc.)

- `int sequential_get_stock(char *product_name, int *stock)`
  - **Description:** gets the stock of a product.
  - **Input:** product name.
  - **Output:** updated stock.
  - **Return value:** 0→ ok, -1 → error (product not found, etc.)

1.1.3. *Concurrent library*

The concurrent library has to be implemented by the students. The different functions must implement the same functionality as their corresponding homologous in the sequential library, as well as the concurrent restrictions previously specified.

- `int concurrent_init()`
  - **Description:** initializes the database and the library resources, including all concurrent mechanisms necessary to protect the access to the database. Use once at the beginning.
  - **Type and scope:** none
  - **Return value:** 0 → ok, -1 → error

- `int concurrent_destroy()`
  - **Description:** frees the resources of the database and the resources used by the library, including all synchronization mechanisms. Use once at the end of the program.
  - **Type and scope:** none.
  - **Return value:** 0 → ok, -1 → error

- `int concurrent_create_product(char *product_name)`
  - **Description:** stores a new product in the warehouse. If the product already exists the operation is considered successful.
  - **Type and scope:** GLOBAL WRITE
  - **Input:** product name.
  - **Return value:** 0 → ok, -1 → error

- `int concurrent_get_num_products(int *num_products)`
  - **Description:** gets the number of products in the warehouse.
  - **Type and scope:** GLOBAL READ
  - **Output:** number of products.
  - **Return value:** 0 → ok, -1 → error

- `int concurrent_delete_product(char *product_name)`

    - **Description:** deletes from the warehouse the first matching product (the whole product including all its stock). If the product is not found the operation is considered successful.
    - **Type and scope:** GLOBAL WRITE
    - **Input:** product name.
    - **Return value:** 0 → ok, -1 → error

- `int concurrent_increment_stock(char *product_name, int stock, int *updated_stock)`
    - **Description:** increments the stock of a product and returns the updated stock.
    - **Type and scope:** GLOBAL READ. LOCAL (to that product) WRITE.
    - **Input:** product name and amount of stock to be added.
    - **Output:** stock updated.
    - **Return value:** 0 → ok, -1 → error (product not found, etc.)

- `int concurrent_decrement_stock(char *product_name, int stock, int *updated_stock)`
    - **Description:** decrements the stock of a product and returns the updated stock.
    - **Type and scope:** GLOBAL READ. LOCAL (to that product) WRITE.
    - **Input:** product name and amount of stock to be subtracted.
    - **Output:** stock updated.
    - **Return value:** 0 → ok, -1 → error (product not found, etc.)

- `int concurrent_get_stock(char *product_name, int *stock)`
    - **Description:** gets the stock of a product.
    - **Type and scope:** GLOBAL READ. LOCAL (to that product) READ.
    - **Input:** product name.
    - **Output:** updated stock.
    - **Return value:** 0 → ok, -1 → error (product not found, etc.)

1.2. *Initial code*

In order to facilitate the realization of this assignment an initial code is provided in the file lab2_concurrency.zip. To extract its contents, you can use the **unzip** command:

```
unzip concurrency.zip
```

As a result, you will find a new directory *concurrency/*, onto which you must develop the lab. Inside this directory you will find:

concurrency/Makefile
File used by the make tool to compile the program. **Do not modify this file.** Use *$ make* to compile the program and *$ make clean* to remove compiled files.

concurrency/lib
Directory which contains the libraries.

concurrency/lib/libdb_warehouse.a
Library that contains the database, compiled for 32 bits.

concurrency/lib/libdb_warehouse-64bits.a
Library that contains the database, compiled for 64 bits. If you want to use this, overwrite the previous file.

concurrency/include
Directory which contains the header files

concurrency/include/db_warehouse.h
Header file of the data warehouse.

concurrency/include/sequential.h
Header file of the sequential library.

concurrency/include/concurrent.h
Header file of the concurrent library.

concurrency/sequential.c

Contains the code of the sequential library. It is provided with the aim of showing an example of how a library can act as a wrapper of the database interface. This library already includes all features requested apart from the synchronization mechanism required to guarantee consistency.

concurrency/concurrent.c

File used by the student as starting point. Complete the code with the synchronization mechanisms. **ONLY MODIFY THIS FILE**.

concurrency/sequential_example.c

Test program which uses the sequential library.

concurrency/concurrent_example.c

Test program which uses the concurrent library for illustrative purposes. Evaluation test will be more complex and will cover more use cases.

2. Assignment submission

2.1. *Grading*

This lab will be graded as follows:

40% - Database scope operations: create, delete, and get number of products. (GLOBAL READS and GLOBAL WRITES).

40% - Product scope operations: increment, decrement, and get stock of a product. (GLOBAL READS, LOCAL READS and LOCAL WRITES).

20% - Memoria.

2.2. *Tests*

The student code must pass the following tests:

- Several threads creating/deleting products simultaneously. Following the restrictions, only one of the threads can perform a global write operation each time.
- Several threads getting the number of products simultaneously. Following the restrictions, up to MAX_READERS threads can perform a global read operation at the same time.
- Several threads creating/deleting products and getting the number of products. In this case, read operations must wait until a write operation is completed, and a write operation must wait until another write/read operation is completed.
- In case of local operations, the restrictions are the same as those specified for global operations, but only concerning one concrete product. I.e. only one thread can perform increments/decrements over the same product at the same time, but different threads can perform increments/decrements over different products. Moreover, remember that local operations also count as a global read.

2.3. *Deadline*

This lab deadline is recommended for 4 weeks.

2.4. *Submission*

The submission must be done separately for the code and the report.

2.5. *Files to be submitted*

The student must submit the code in a zip compressed file with name AAAAAAAAA_BBBBBBBBB.zip where A…A and B…B are the student identification numbers of the group. A maximum of 2 persons is allowed per group, but it can be individual. The file to be submitted must contain:

- **concurrent.c**
- **authors.txt**

Authors file must contain the name and the name of each author in separated lines:

AUTHOR1

AUTHOR2

The lab report must be submitted in a PDF file. A minimum report must contain:

- **Description of the code** detailing the main functions it is composed of. Do not include any source code in the report.
- **Tests cases** used and the obtained results. All test cases must be accompanied by a description with the motivation behind the tests. In this respect, there are three clarifications to take into account.
    - Avoid duplicated tests that target the same code paths with equivalent input parameters.
    - Passing a single test does guarantee the maximum marks. This section will be marked according to the level of test coverage of each program, nor the number of tests per program.
    - Compiling without warnings does not guarantee that the program fulfills the requirements.
- **Conclusions**, describing the main problems found and how they have been solved. Additionally, you can include any personal conclusions from the realization of this assignment.

Additionally, marks will be given attending to the quality of the report. Consider that a minimum report:

- Must contain a title page with the name of the authors and their student identification numbers.
- Must contain an index.
- Every page except the title page must be numbered.
- Text must be justified.

Do not neglect the quality of the report as it is a significant part of the grade of each assignment.

3. Bibliography

- El lenguaje de programación C: diseño e implementación de programas Félix García, Jesús Carretero, Javier Fernández y Alejandro Calderón. Prentice-Hall, 2002.
- The UNIX System S.R. Bourne Addison-Wesley, 1983.
- Advanced UNIX Programming M.J. Rochkind Prentice-Hall, 1985.
- Sistemas Operativos: Una visión aplicada Jesús Carretero, Félix García, Pedro de Miguel y Fernando Pérez. McGraw-Hill, 2001.
- Programming Utilities and Libraries SUN Microsystems, 1990.
- Unix man pages (`man function`)