CSE 320 Spring 2019 HW #5

April 29, 2019

Deadline: May 12, 23:59 Stony Brook time (Eastern Time Zone).

1 Introduction

The goal is of this homework is to expose you to the various topics covered in the class. This homework will help you better understand topics such as caches and virtual memory. You will implement a simple program that uses such concepts as virtual address translation, page tables, and caches. You will also use bits of knowledge from the previous assignments.

As usual, we highly recommend spending some time thinking about your code. You may want to design your code first, think about possible problems, and probably even write it down on a piece of paper. It may significantly reduce the amount of time that you will spend on writing the code itself.

2 The homework

This homework consists of three parts. Please read following subsections to find more details about each part.

In this homework, you will need to implement a simple client-server system. It uses concepts that you saw throughout the course but focuses on virtual memory and caches. The first step will be to create two programs. The first one is a client (think process) that connects to some remote database (think memory) to perform some operations on it. The second program is a remote database (think memory) that can have multiple clients to be connected to it and needs to be able to distinguish them and handle their requests. Then later in the homework, you will optimize that system by using a caching mechanism, which is common not only in your CPU but also in other applications (think CDNs, memcached, etc).

2.1 Part I

This part is traditionally about readings that you may find useful for this homework. The most important one is from the textbook, specifically, chapters 6 and 9.

Also, the following links on inter-process communication and address translation may be helpful:

IPC
Named pipes
Address translation

2.2 Part II

In this part, your task is to write a simple art database client-server system. Using your client user will connect to the remote database to perform operations on it. You need to implement at least the two following files described further in the text:

- client.c - this is a client part of the database system.
- warehouse_db.c - this application replicates remote database system.

More details can be found below.

2.2.1 client.c

This is the client application, which provides an interface to communicate with the remote database system. It should support the following commands:

- start - starts a new session with remote database.
- alloc - adds a new "art entry" to the list of art entries.
- dealloc *ID* - deletes an art entry with a specified ID from the list of art entries.
- read *ID* - prints art name that corresponds to an art entry with a specified ID.
- store *Y Z* - stores art *Z* in art entry *Y*.
- close - closes session with the remote database.
- infotab - prints information about translation tables.
- exit - clean everything and exit application.

Now let's discuss all commands in greater details:

start. This command starts a session with the remote database. It should spawn a new thread and communicate its thread ID to the remote database. Once the session is started, the user should not be able to start new sessions unless the previous one is closed. The thread should keep running as long as the session is still up.

alloc. This command adds a new art entry to the list of art entries. Thus, your program should maintain the list of entries and they are on a per-client basis. Initially, the list is empty. The local art entries should not contain art names as they should be stored only in the remote database. It also means that the client should reserve a record entry in the remote database.

dealloc *ID*. This command should remove the entry from the list of art entries. It also should free (mark invalid, see later) the corresponding record entry in the remote database.

read *ID*. This command should print an art name stored in the art entry with a provided ID. The client will issue a request to the remote database to retrieve the art name.

store *ID Z*. This command sets an art name to Z in a specified art entry. This information should be propagated to the remote database to store that information.

close. This command should close the session. It also should free all art entries and anything related to this session (including records in the remote database).

exit. This command should exit the program.

When you establish a session, your client program should create a set of tables (think two-level page tables) to perform translation between art entry ID and ID in the remote database. The art entry ID on the client side is one byte and have the following format:

- First 2 bits - reserved for future use (i.e., currently unused).
- Second 4 bits - index in the first table.
- Last 2 bits - index in the second table.

Soon you will see more information about records in the remote database, but the idea here is that you need to translate local art entry ID into record ID in the remote database.

infotab. This is am interactive command that helps the user to navigate the translation tables content. Initially, it should print the first level tables and prompt user to choose one of these tables. Then the user should be able to navigate from that first level table to the second level table. Printed information should be clear enough to perform manual ID translation.

There can be at most four clients at any given time.

2.2.2 warehouse_db.c

This program emulates the remote warehouse database. Clients will connect to the database to communicate with it. Also, to emulate the database being remote, it should sleep for one second before handling any received message. By default, this program does not have a shell, however, if a user sends the `SIGINT` signal to this program then your program should catch that signal and enable shell. After handling the command from the user, the shell should again disappear. The following commands should be supported in the shell:

- list - lists current art managers (clients) along with their IDs.
- list *X* - lists art entries used by "art manager" *X*.
- dump - prints information about all records in the database.
- exit - exits the program.

list. This command prints all established sessions (connected clients) with their IDs.

list *ID*. This command prints all entries used by the client with a specified ID.

dump. This command prints information about all records in the database. Information should include an ID of the record, stored value (art name), an ID of the client (if any) who is the owner of the record, and valid bit.

exit. This command exits the program.

This is a separate application that should be compiled and executed separately from the client programs.

This program takes one argument from the CLI that will specify the number of records that the database can store. For example, if the program is started with argument *100*, then it can store up to one hundred records. If there are *N* records, then all records have index from *0* to *N - 1*. Furthermore, each record should be able to store the art name (not more than 255 characters), an ID of the client, and a valid bit that indicates that the record is currently in use. If the record is not in use, then that record can be reclaimed by other clients.

If there are any messages received from clients or the remote database sends messages to the clients, then this program needs to print some respective message to the output. For example, if client *115* writes an art name *"Starry Night"* to the record *7*, then database can print something similar to the following:

```
Client 115 set record 7 to ‘‘Starry Night’’
```

This is just an example, but it should give you an idea of how detailed message should be. You also need to print these log messages using colors that are different from the default one. Each client should have own color. As you have up to four clients, you can implement a round-robin system to choose which color should be assigned to a particular client.

2.2.3 Communication between client and DB

It is up to you what mechanism to use for the communication between clients and remote database. We suggest using named pipes (FIFOs). As we limited how many clients there can be at any given time, you can create four FIFOs and use them for communication. It may be a good idea to pass the name of the FIFO through command-line arguments during the client start.

As you may notice, we mentioned some session establishment (`start` command). Your client should send a message to the server to establish a connection. It is up to you to come up with any simple protocol for that. For example, a client may send a message "connect X" where X is the client's ID and the remote database may respond "OK" if everything is good.

Also, every time you need to send a message from the client to the remote database you need to be able to distinguish between different clients to protect records of one client from the other. You also can assume that once the client disconnects, the data of that client is not valid anymore.

2.3 Part III

For Part III you need to add a simple caching mechanism to your client. Your cache should use remote database record ID (think physical address in the real cache). Thus, when you have remote database access you need first to translate local art entry ID into remote record ID. Your cache can contain at most four values. You can use any replacement policy you want. You also free to implement any type of cache though we recommend to do direct mapped cache as it is the simplest and the most straightforward. When you receive a remote database access request your client first should go into the cache and then there are several situations possible:

- cache hit - you should print "**cache hit**" and return that art name to the client, thus avoiding remote database access.
- cache miss - you should print "**cache miss**" and search for the art name in the remote database. After you found that value and if there are no errors you should put that value into your cache and return it to the client.
- line eviction - when your cache is full and you need to add another line you should evict one of the current cache lines. In case that happens, you should print "**eviction**".

2.4 EC

There are several EC for this homework. You will need to schedule an appointment with me to demo your EC. Deadline for the EC is the last lecture of this semester.

2.4.1 Multiple Clients

Currently, we need to start multiple client applications to emulate having multiple clients. In this extra credit, you need to modify your program to allow a user to switch between multiple client sessions within one program. Mind that in such case you need to switch translation tables either.

2.4.2 Many Clients

Currently, our client-server system supports only up to four clients. You need to modify that behavior to support the arbitrary amount of clients. You may find sockets being useful for this extra credit.

2.4.3 L2 Cache

Currently, we have only one level of cache. In this extra credit, you need to add second level cache that can store up to eight values.

2.4.4 Networking Programming

This extra credit is substantially bigger than the previous ones. It is in a way promised extra homework on the network programming. Your goal here is to rewrite this homework but instead of running everything in one virtual machine, you need to start multiple virtual machines. One VM will be running the database while other VMs (no limit on their number) will be clients that can connect to the database. You can hardcode the IP address of the database VM and you will need to account for possible disconnections to preserve data in the database and allow reconnects. If you will decide to implement this extra homework then contact me directly so I can provide you some guidance and refer to the readings that may help.

2.5 GitHub Link

Please click on the link below that will set up a repository for you for this homework. In case, it will take more than a few minutes please contact me so I can set up the repository for you manually.

https://classroom.github.com/a/XWUkQB38

2.6 Requirements

If there is an error or wrong command was entered then you need to print on screen textual description of that error. There is no strict requirement on the text but make that text self-explanatory.

Also, your program should not be stuck or crash and this time there should be no memory leaks and errors (including thread related).

You need to create a `README` file similar to the previous homework (see below).

2.7 Submission

As before, we will grade the latest push. Similar to the previous homework, you need to create a `README` file that will clearly state how to compile and run your code. It also should provide high-level details about your program and your code. Please also make sure that `make` will compile all the programs in your submission.