Library management system design

Contents

1. Introduction

This document describes a library management system design. During design stage, I like to use top-down approach. First, understand the scope of the system, fully understand the functions the system need to provide. Then divide the function into may logical smaller modules. For each smaller modules we do the same procedure. By this divide and conquer approach, we can split a big problem into easier to solve, smaller pieces. When start to develop the system, I usually do it top down or bottom up. In the license server project, for example, I used top down and test driven development approach to code the system.

The library management system is designed to help customers easily access the library resources, and to help library employees easily management the library. In the process of the object oriented system design, the following points are considered:

1. The scope of the system, i.e. what's the basic functionality for this system.
2. What frame work can be used to this system? The system should be designed to be extensible and easy to be upgraded.
3. What design patterns can be used during class designs?
4. System scalability. How to make sure the system is efficient and scalable.
5. How to design the system to make it easier to be tested and debugged.

In section 2, I will further illustrate the considerations for this system design. In section 3 and 4, I will introduce the class designs for the different entities in the library system. Section 5 describes the thread and thread pool design. Finally I will conclude this document in section 6.

2. System design considerations

The scope of the library system is designed to manage a big library with many branches. There can be thousands of customers in the system. Basically, the following scenarios should be supported.

- There can be thousands of customers for the system. Users can register, login, logout system, can search for medias (book, videos, etc) from any branch. They can reserve medias, borrow medias. They can check their account status, for example, they can check their checked out media list; they can pay fines charged for their late return.
- There are employee accounts for the library. Each employee has a role, i.e. manager, system administrator, operator, etc. Different role has different privilege. For example, manager can check all employee's (of their branch) account status; system administrator can change employee's password, etc.
- There are media types. The media can be books or videos.

- Section 3 will cover the entity data class design in detail.

It makes sense to design the whole system by using Model-View-Control framework. The model part is the data of the system, including the media data, employee account data and customers account data, transaction data, etc. The view part is the classes of the front end GUI class design. The GUI can be a webpage, or a GUI application. I will not cover this part for now because of the time limit. The control part takes care of all the transactions of the system, for example, when user try to borrow a book, the control module need to make sure that user's account is in good shape to be allowed to borrow books.

In the class designs, I also keep in mind to use available OO design patterns to make sure the design is extensible and easy to be understood. For example, it makes sense to use observer pattern to design the notice system. Many customers can register himself as a observer to a book, when the book is available an notice email can be sent to the first waiting the observer user. So she can go to library to borrow the book.

Because the system is designed to be used by thousands of users, the scalability is very important. For example, we should make sure the system respond to any user's request in no more than 3 seconds. In order to make it extensible, we need to use many data structures, for example, hash table, cache, etc.

3. Classes for library entity data

3.1 Locks

The library management system is a distributed system. There can be hundreds of or thousands of threads to handle user's request. We need to design a uniform lock class to help protect critical data.

```
Class Lock {
public:
        void lock();
        void unlock();
private:
        // lock data.
};
```

Sometimes, this class is very useful:

```
class SmartLock {
        SmartLock(Lock &lk) : theLock(lk) {
                theLock.lock();
        }
        ~ SmartLock() {
                theLock.unlock();
        }
private:
        Lock &theLock;
};
```

We lock a lock in it's constructor and unlock the lock in it's destructor.

3.2 Customer Account class

```
Class Address {
public:
        Address(const string &stNum, const string &stName, const string &city, const sting &stateId);
        ~ Address();

        const string getStreetNumber() const;
        void setStreeNumber(string number);
        // other get/set functions….

        bool operator==(const Address &adr) const; // test if the two address the same
private:
        string streetNumber;
        string streetName;
        string cityName;
        string stateId;
};

Class AccountState {
public:
        enum  STATE{ACCOUNT_VALID, ACCOUNT_SUSPENDED, ACCOUNT_EXPIRED,
                ACCOUNT_NEED_UPDATE_INFORMATION};
public:
        AccountState(STATE st);
        ~ AccountState();

        bool isValid() const;
        bool isSuspended() cost;
        //… other definitions here.
        // define the account state related
};

Class Account {
public:
        int getAccountId() const;
        void setAccountId(const int id);
        string getAccountName() const;
        void setAccountName(const string &name);
        string getAccountPassword() const;
        string getAccountPassword();
        Address *getUserAddress() const;
        void setUserAddress(const Address &address);
        string getPhoneNumber() const;
        void setPhoneNumber(const string &phone);
        string getEmailAddress() const;
        void setEmailAddress(const string &email);
```

```
        AccountState &getAccontState(); // get the reference of the state.
        Lock &getAccountLock();
protected:
        int customoerID;
        string accountName;
        string password; // account password
        string FirstName;
        string MiddleName;
        String LastName;
        Address *address; // Use composite design pattern here.
        string emailAddress;
        string phoneNumber;
        AccountState state; // the account state: valid, suspended, etc.
        LogInRecord lastLogInRecord; //

        Lock      accountLock; // lock to protect this account
};

class Customer: public Account {
public:
        Account(string id, string name, string password);
        ~ Account();
private:
        list<MediaData *> borrowedMedia;
        list< MediaData *> reservedMedia;
        double  fineCharge; // the fined money because of late return or lost media
};
```

Basically, we design an account class to keep the common data information for customer account and library employee account data. The customer account specific data, for example, the checked media list, are kept with the customer account itself.

For each account, we keep the account details, for example, the account id number, the account name, etc.

3.3 Library Branch Data Class

We also have library branch data class to record the library branches.

```
Class LibraryBranch {
public:
        LibraryBranch(string id, string name);
        ~ LibraryBranch();

        string getBranchId() const;
        void setBranchId(const string &id);
        string getBranchName() const;
```

```
        void setBranchName(const string &name);
        Address *getBranchAddress();
private:
        string branchId;
        string branchName;
        Address *address;
};
```

3.4 Library employee Account class

```
class Employee: public Account {
public:
        enum  EmployeeType{
                EMPLOYEE_MANAGER,
                EMPLOYEE_SYS_ADMIN,
                EMPLOYEE_OPERATOR
        };
public:
        Employee();
        ~ Employee();
        //…… (skip the definition here)
private:
        EmployeeType type;
        string workPhoneNumber;
        string extensionNumber;
        string branchId; // Assume a employee can only work for one branch
};
```

3.5 Media class

```
Class MediaSaveLocation {
public:
        MediaSaveLocation(string branchId, int rowNum, int columnNum);
        ~ MediaSaveLocation();
private:
        LibraryBranch branch; // in which library branch
        int rowNumber;
        int columnNumber;  // row number and column number gives the location in the branch.
};

Class Media {
public:
        virtual bool isBook() const { return false;}
        virtual bool isVideo() const { return false;}
        virtual bool isAudio() const { return false;}
```

```
        Media();
        ~ Media();

        Lock &getMediaLock();
private:
        string id; // the unique id number for a media
        MediaSaveLocation location; // the location for this media
        long      checkedOutTimeSpan; // how long this item can be check out (in days)
        List< Account *> observer; // the observer design pattern.
        Lock mediaLock; // to protect this media in multi-thread environment
};


Class BookMedia:public Media {
public:
        virtual bool isBook() const { return true; } // this is a book media
        BookMedia(); // no parameters here. Need
        ~ BookMedia();

        // get/set functions for private data members
private:
        string title;
        vector<string> authors;
        string ISBN;
        string publisher;
        string publishTime; // year/month..
        int pages;
        string Description;
        // … others…
};
```

Definitions for VideoMedia and AudioMedia class are similar as BookMedia.

3.6 Transaction Data class

When we have defined account class, we also need to define transaction data class. Transactions includes:

- Customer Borrowed medias
- Customer returned medias
- Customer paied charged money
- Customer changed account settings
- Customer started a media searched

- Employee changed media storage location
- etc….

Here we can use Command design pattern. Basically we design the transactions as commands. So it is easy to redo/undo the transactions.

```
Class TransactionCommand {
        enum CommandType{
                TYPE_RETURN_MEDIA, TYPE_CHECKED_OUT_MEDIA,
                TYPE_CHANGE_ACCOUNT_SETTING,
                TYPE_PAIED_MONEY,
                TYPE_SEARCH_MEDIA,
                TYPE_MODIFY_MEDIA_LOCATION,
                // other definitions
        };
public:
        TransactionCommand(const string &accountId, const string &ipAddress, CommandType type);
        ~ TransactionCommand();
private:
        Account *account; // who did this transaction
        string    transactionId; // each transaction has a unique id in database
        time_t  transactionTime; // when this transaction happened
        stirng    ipAddress; // the ip address from which this transaction was initiated
};
```

Note: In TransactionCommand, we used Account *account to note who did this transaction. We are not using the account id because when we try to locate the account data from account id, there are too much resource involved. We can not make the system scalable. We are using Flyweight design pattern here, so we can use the object data efficiently.

```
class CheckedOutMediaTransaction: public TransactionCommand {
public:
        CheckedOutMediaTransaction();
        ~ CheckedOutMediaTransaction();
private:
        Media * media;
};
```

Here we also use Media pointer itself, also Flyweight design pattern, to make the system more efficient.

Other transaction class definitions are similar.

4. Classes for library management functions

Section 2 and 3 defined the Model part of the Model-View-Control framework. This section defines the control part. This part need to accept the request data from the View part and translate the request to

commands, doing necessary validity checking and apply the request to the model data. By doing this, we separated the management of the three part, so the system can be better separated into modules. Then the whole system can be divided into pieces and assign each piece to developers.

In order to make the system scalable, we need to handle big data. We can use some data structure, for example cache, hash table, hash map, etc, to make the data accessing faster. Map reduce techniques can also be used in some scenario, for example, when we want to count for the available copies of each media, we can use the following approach:

Assume we have 100 computers to do the calculation. We do:

(1). For each media, calculate a number m = hash(media name) % 100, let the machine m query the Database and calculate the available copy for each media

(2). Each machine send its own result to a centralized machine

(3). The centralized machine will collect the machine and reduce the result to get the final result.

4.1 A LRU cache design

We will need to use cache to accelerate data access. For example, when people search for a kind of media, it is highly possible that after he get the result from a seach, he may add more keyword to try to get fewer but more accurate result. It makes sense to cache his previous search result. When we see a new search, we can check if the keywords in the new search contains the keywords of a previous search. If we can find such a previous search, we can try to get the more accurate search result from the previous search result, instead of querying the database again. The LRU cache can be designed as follows:

```
Template<class KeyType, class ValueType>
Class LRUCache {
public:
        bool canHit(const KeyType &key) const;
        class ValueType getValue(const KeyType &key)  const;
        void addNewCache(const keyType &key, const ValueType &value);
private:
        list<pair<KeyType, ValueType> > data; // the double linked list
        unordered_map< KeyType, list<pair<KeyType, ValueType> >::iterator> valueMap;
}
```

Here we have a LRU cache for KeyType and ValueType. For example, for the search scenario, the KeyType can be vector<string>, which is a list of keywords, and the ValueType can be list<Media *>, which is a list of Media.

The double linked list assures that we can remove/insert/move a node to front in O(1) time. The valueMap assures that we can locate a value from a key in O(1) time.

4.2 Hash map

We may also need hash map in the system. For example, we may need to look for an account from it's account id. If we do the look up from database, or from a linked list in memory, every time, the cost will be too high. By using hash table, we can locate the account in O(1) time.

In C++, we can use the unordered_map<sting, Account *> directly.

4.3 ActionCheckedOutMedia class

In the design of library management functions, each class is actually is an action. For example, in stock management function, we may have actions: add a new media, delete a media, change media location, etc. In 4.3, I want to use an example, check out media management, to describe my thinking in this type of Control part design.

When we want to perform an action, we need to do several steps:

1. Check if the action is valid. For example, the account is allowed to do this action or not? The media is a valid media? There is enough media copies for this action? ..

2. We need to do the action in a transaction, i.e. either the action is finished successfully or the action failed completely. There should be no intermediate state for any action.

```
class ActionCheckedOutMedia {
public:
        ActionCheckedOutMedia(Account *account, Media *media);
        ~ ActionCheckedOutMedia();

        void commitAction();

        void undoAction();
        void redoAction();

private:
        // …
};
```

5. Thread processing

The library management system is a distributed system which supports thousands of users, including customers, employees, etc. We must have hundreds of or even thousands of threads to server each request. The License system I designed and implemented in our company has a problem, which is not critical now, but will be a problem later when people run thousands of simulations. I always ask myself how I can improve it, and if I redesign the system what new approaches I will use to make the license system scalable.

Come back to the library management system, in previous sections, I jump into the low level detail and designed the account data class, media data class, and the transaction data class. From a top level, we can look the whole system as composed of thousands of threads which are handling user's request. The data structure and algorithm for the threads are critical for the whole system to perform well. In order to make the system scalable, we need to consider:

- Use thread pool.
- Prioritize user's request

5.1 Thread pool and thread class

The thread pool class

```
class ThreadPool {
        ThreadPool(int minMainTrheadNumber,
                        int maxMainTrheadNumber,
                        int minWorkerThreadNumber,
                        int maxWorkerThreadNumber,
                        );
        ~ThreadPool();

        void createANewMainThread();
        void createANewWorkerThread();
        void destroyAFreeMainThread();
        void destroyAWorkerTrhead();

        Thread *getAFreeWorkerThread();
        Thread* getAFreeMainThread();

private:
        int minMainTrheadNumber;
        int maxMainTrheadNumber;
        int minWorkerThreadNumber;
        int maxWorkerThreadNumber;

        Lock workerThreadListLock;
        list<WorkerThread *> freeWorkerThreadList;
        list<WorkerThread *> runningWorkerThreadList;

        Lock mainThreadListLock;
        list<MainThread *> freeMainThreadList;
        list<MainThread *> runningMainTrheadList;
};
```

Therecan be one to a hundread mainthread running in the system. The only purpose for a main thread is to accept the user's request and put the request in a priority queue. It will pick the highest priority

request to a worker thread to let the worker thread further handle the user's request. A main thread should finish all the tasks in no more than 100 millisecond. Then it will switch to sleeping state for further user connection. Usually a couple of main thread is enough for smaller system. But for big system, and there are thousands of connections coming, we have to create more main thread.

The worker threads are the entities to handle user's request. It may do the library media search, may changing account setting for system admin account, etc. We require the response time for a worker thread should be no more than 3 seconds. More than 3 seconds will cause a bad user experience. There can be more than thousands of threads in system.

```
class Thread {
public:
	Thread();
	~ Thread();

	virtual void start()=0; // pure virtual function here

	const &ThreadState getThreadState() const;
	// For a thread, we want to know how much time it spends on a state;
	// For example, when processing a request, if more than 3 seconds the thread did not send
	// back result data to client side, there may be problems in the processing
	unsigned long getTimeOnState(const ThreadState &state) const;
private:
	// There is a state machine for the thread status, like the process state in Linux kernel design
	ThreadState state;
	unsigned long threadId;

};

class MainThread {
	void setUserConnection(const ConnectionType &conn);
	void start();

private:
	ConnectionType userConnection;
};

class WorkerThread {
	void setUserRequest(const RequestType &req);
	void start();
private:
	RequestType userRequest;
};
```

We can design further detailed thread class for worker thread. For example, we can have MediaSearchWorkerThread, AccountUpdateWorkerThread, etc.

5.2 Prioritized requests

We have mentioned that we can use command design pattern to organize the user request handling. By using command design pattern, we can easily create transactions for requests, we can undo/redo operations.

The requests should be prioritized. For example, when user search for a book, he/she may expect there will be a little bit delay because he/she knows the system need to find books from huge amount of data in database. But when user is checking out borrowed book online, he/she may not want to tolerate any delay. In the requests design, we can give higher priority to book checkout request than to book search request. By using priority queue, we can prioritize the user requests.

I skip the further discussion of prioritized request here.

6. Conclusion

In this document, I described my design for the library management system. This system can be a huge system. I only covered a tiny piece of it. I tried to use design patterns to make the system extensible. I also used cache, hash map data structure and map reduce techniques to make the system scalable. The system can be implemented in Model-View –Control frame work. The view part design is not covered in this document. By separate the whole system into many modules, it is easy to divide the system into smaller pieces.

In designing a system, it is also important to design the test/debug strategies. Actually we can apply the agile approaches in developing the system. The test driven development enables use to automate the tests, so a lot of bugs can be automatically detected.

Thank you very much for your time for my interview and reading this document.

Have a nice day!