Relazione sulla tesina del corso di Sistemi Operativi (AA 2018/2019)

Fabio Buracchi, 0253822

PREFAZIONE

Nella realizzazione del progetto è stato utilizzato esclusivamente codice scritto dall'autore, presente nelle librerie standard del C, presente nella versione Win32 delle Windows API e nelle API dello standard POSIX.

La soluzione raggiunta si compone di tre programmi:

- cinemad, adibito alle mansioni di server
- cinemactl, un server management client destinato all'amministrazione del server in locale
- cinema-client, un thin client destinato all'utenza esterna

Durante il corso di questa relazione ci si riferirà con l'appellativo di client unicamente all'applicativo cinema-client mentre per riferirsi al cinemactl si farà sempre riferimento al suo scopo di strumento di amministrazione.

Specifica

Realizzazione di un servizio di prenotazione posti per una sala cinematografica, supportato da un server.

Ciascun posto è caratterizzato da un numero di fila, un numero di poltrona, e può essere libero o occupato. Il server accetta e processa sequenzialmente o in concorrenza (a scelta) le richieste di prenotazione di posti dei client (residenti, in generale, su macchine diverse).

Un client deve fornire ad un utente le seguenti funzioni:

Visualizzare la mappa dei posti in modo da individuare quelli ancora

disponibili.

1. Inviare al server l'elenco dei posti che si intende prenotare (ciascun posto da prenotare viene ancora identificato tramite numero di fila e numero di poltrona).
2. Attendere dal server la conferma di effettuata prenotazione ed un codice univoco di prenotazione.
3. Disdire una prenotazione per cui si possiede un codice.

Si precisa che la specifica richiede la realizzazione sia dell'applicazione client che di quella server.

Per progetti misti Unix/Windows è a scelta quale delle due applicazioni sviluppare per uno dei due sistemi.

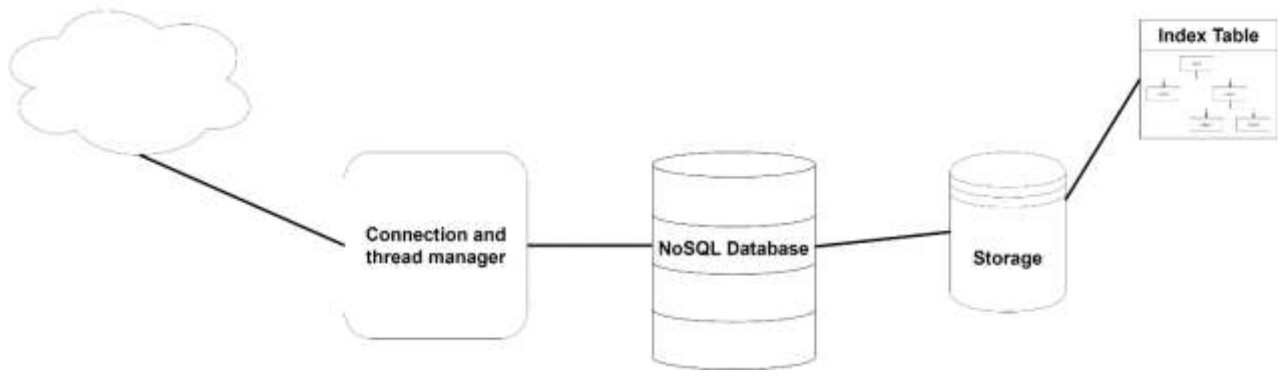
Scelte implementative

È stato deciso di sviluppare il client per sistemi Windows in modo tale da poter sviluppare in maniera esaustiva il tema di finestra come argomento del corso, mentre il server è stato realizzato in ambiente Unix like scegliendo di far ad esso processare in concorrenza le richieste dei client, sviluppando così un software multithreaded.

Il client si compone di un modulo principale e di uno ausiliario per la gestione dei dati in persistenza con cui tratta il codice di prenotazione univoco. Nel modulo principale oltre a venir registrata la classe della finestra principale e ad elaborare i messaggi ad essa destinati, vengono utilizzate delle bitmap per poter rappresentare graficamente lo stato dei posti ed un timer il cui segnale richiama la procedura per l'aggiornamento della mappa dei posti.

Il client destinato all'amministrazione permette di eseguire operazioni quali l'avvio del server, lo spegnimento, il controllo dello stato e l'invio di query.

Il server si compone di un modulo principale adibito alla gestione delle connessioni e dei thread, un modulo che implementa un semplice database NoSQL, uno per la gestione dell'IO sul dispositivo ed un ultimo modulo per la gestione delle informazioni dei dati in memoria su disco.



Multithreading

Il server come già anticipato è un'applicazione multithread, nello specifico nel modulo destinato alla gestione dei thread e delle connessioni i thread si dividono in cinque categorie:

- Main thread, dopo aver inizializzato le risorse e gli altri thread si pone in attesa del segnale SIGTERM e, una volta ricevuto, gestisce la corretta chiusura dell'applicativo.
- Connection manager, è un tipo di thread che in un loop infinito accetta le richieste di connessione sul socket in ascolto ad esso associato, generando per ciascuna di esse un thread per evadere le corrispondenti richieste
- Request Handler, è un tipo di thread che crea, creato un thread ti tipo timeout, tenta di ricevere la richiesta del client, in caso positivo invia un segnale di chiusura al timer thread, evade la richiesta ed invia il risultato al client.
- Timeout thread, è un tipo di thread che invia un segnale di chiusura al thread padre dopo un dato intervallo di tempo
- Joiner thread, è il thread che si occupa di liberare la memoria allocata da un thread attraverso l'operazione di join, attraverso il suo utilizzo i thread Request Handler hanno la garanzia di poter terminare l'evasione di una data richiesta senza essere interrotti in fase di terminazione.

Connessioni

Per quanto riguarda le connessioni, necessitando di comunicazioni orientate alla connessione, sono stati utilizzati socket AF_INET via IPV4 e TCP/IP tra il server ed il client ed un socket AF_UNIX per la comunicazione tra il server ed il client destinato all'amministrazione.

È stato realizzato un apposito modulo compatibile con gli standard POSIX/Win32 per la gestione delle connessioni in modo tale da semplificare l'attività di codifica.

Database, procedure e gestione della concorrenza

Il database NoSQL è un semplice database chiave valore. Non presenta un componente per la gestione dello scheduling delle operazioni in quanto è stato scelto di utilizzare il modulo storage per il mantenimento dei dati che rendono possibile la concorrenza.

Sebbene questa scelta aumenti il quantitativo di memoria richiesto dall'applicazione a run-time e diminuisca il livello di granularità con cui è possibile gestire la concorrenza (ad esempio non è possibile ottenere un lock di predicato) per il dominio del problema, che impone un piccolo quantitativo di informazioni da dover memorizzare in persistenza ed una gestione poco più che rudimentale della concorrenza, attraverso questa scelta implementativa è stato possibile ridurre il livello di complessità dell'applicativo.

L'API principe di questo modulo è database_execute che, ricevendo in ingresso una query sottoforma di stringa ed un puntatore dove poter registrare il risultato, evade la richiesta attraverso una delle procedure codificate all'interno del modulo.

In particolare durante l'esecuzione di richieste di tipo get, gestite dalla procedura procedure_get si richiede un lock di tipo condiviso sul dato in esame prima di effettuare l'operazione di lettura, mentre durante le richieste di tipo set, gestite dalla procedura procedure_set si richiede un lock di tipo esclusivo sul dato in esame prima di effettuare l'operazione di scrittura.

Per quanto riguarda la prenotazione, gestita della procedura procedure_book, viene effettuato un lock a due fasi per garantire la correttezza dell'operazione.

Storage, index table e tipologie di semafori

Il componente storage permette l'utilizzo dei dati in persistenza attraverso uno stream verso il file in cui risiedono i dati ed un'area di memoria che contiene una copia delle informazioni su disco che permette di aumentare il livello di multiprogrammazione e ridurre il numero di canali aperti per le operazioni di lettura e scrittura, essendo giustificata dalla quantità esigua di dati da dover mantenere in memoria secondaria per la natura del problema.

Le scritture sul file utilizzano il mutex mutex_seek_stream per rendere atomiche le operazioni di seek e scrittura attraverso lo stream, le letture utilizzano invece il lock condiviso lock_buffer_cahce di tipo rwlock che bloccano in modalità shared attraverso la funzione pthread_rwlock_rdlock per leggere dall'area di memoria.

Le operazioni di scrittura che aggiungono nuove chiavi in memoria, sebbene assai rare sono anche particolarmente costose, in quanto oltre alla semplice scrittura hanno l’onere di aggiornare, dopo l’operazione `pthread_rwlock_wrlock`, l’area di memoria che funge da cache.

Il lock sui singoli elementi chiave valore è possibile attraverso il mantenimento in memoria di un lock di tipo rwlock per ciascuno di essi.

La posizione del dato su disco ed il lock ad esso associato costituiscono la coppia di valori mantenuta in memoria principale nella index table, in particolare la index table è implementata come un albero AVL che mantiene le chiavi organizzate secondo un ordinamento lessicografico.

Conclusioni

Sebbene il database NoSQL sia forse troppo rudimentale, privo sia di un modulo dedicato allo scheduling delle operazioni, sia di uno dedicato alla sicurezza, per la natura accademica della specifica si è pensato che la completezza del database e lo sviluppo di argomenti non strettamente legati al corso non fossero parte fondamentale dell’elaborato.

I risultati sperimentali di utilizzo degli applicativi sono confacenti alle aspettative legate agli aspetti teorici con cui essi sono stati realizzati.

Manuale d’uso

Manuale del client:

Il client si avvale di un’interfaccia grafica per permettere all’utente d’interagire con il sistema.

È possibile visualizzare nell’angolo in alto a sinistra della schermata principale in titolo del film e l’orario di riproduzione dello spettacolo che si intende prenotare.

È possibile visualizzare centrato nella parte superiore della finestra principale il proprio codice identificativo.

È possibile interagire con la mappa dei posti attraverso il click del mouse sull’icona di un posto:

- Il click su un posto non disponibile (colorato di grigio) verrà ignorato
- Il click su un posto disponibile (colorato di nero) lo renderà selezionato per la prenotazione (colorato di blu)
- Il clock su un posto prenotato (colorato di verde) lo renderà selezionato per la cancellazione (colorato di rosso)

È possibile ripristinare lo stato originale di un posto con il quale si è interagito attraverso un ulteriore click.

È possibile prenotare o eliminare la prenotazione un posto selezionato per la prenotazione o per la cancellazione facendo click sul pulsante Prenota o sul pulsante Modifica prenotazione.

È possibile eliminare tutti i posti prenotati facendo click sul pulsante Elimina prenotazione e confermando l’operazione.

Manuale del server:

Trovandosi nella dierctory che contiene il file cinemactl:

- per avviare il server eseguire il comando cinemactl start
- per spegnere il server eseguire il comando cinemactl stop
- per visualizzare lo stato del server eseguire il comando cinemactl status
- per eseguire query sul server eseguire il comando cinemactl query "[QUERY]"

Manuale di compilazione:

È stato preparato un makefile per la compilazione, è possibile eseguire il comando:

- make per generare gli eseguibili del client manager e del server
- make clean per eliminare i file ausiliari generati al momento della compilazione
- make remove per eliminare tutti i file generati durante la compilazione per l’esecuzione dell’applicativo

Codice sorgente

resources.h

```c
#pragma once

#include <stdarg.h>

#ifdef _WIN32
#include <Windows.h>
#endif

#ifdef _WIN32
int asprintf(LPTSTR* str, LPCTSTR format, ...);
#elif __unix__
int asprintf(char** str, const char* format, ...);
#endif

int strtoi(char* str, int* result);
```

resources.c

```c
#include "resources.h"

#include <stdlib.h>
#include <limits.h>
#include <stdio.h>
#include <errno.h>

#ifdef _WIN32
#include <Windows.h>
#include <tchar.h>
#endif

#ifdef _WIN32
int vasprintf(LPTSTR* str, LPCTSTR format, va_list args){
	size_t size;
	LPTSTR buff;
	va_list tmp;
	va_copy(tmp, args);
	size = _vsctprintf(format, tmp);
	va_end(tmp);
	if (size == -1){
		return -1;
	}
	if ((buff = (LPTSTR)malloc(sizeof(TCHAR) * (size + 1))) == NULL){
		return -1;
	}
	size = _vstprintf_s(buff, size + 1, format, args);
	*str = buff;
	return (int)size;
}
#elif __unix__
int vasprintf(char** str, const char* format, va_list args) {
	int size;
	char* buff;
	va_list tmp;
	va_copy(tmp, args);
	size = vsnprintf(NULL, 0, format, tmp);
	va_end(tmp);
	if (size == -1) {
		return -1;
	}
	if ((buff = (char*)malloc(sizeof(char) * (size_t)(size + 1))) == NULL) {
		return -1;
	}
	size = vsprintf(buff, format, args);
	*str = buff;
	return size;
}
#endif

#ifdef _WIN32
int asprintf(LPTSTR* str, LPCTSTR format, ...){
	size_t size;
	va_list args;
	va_start(args, format);
	size = vasprintf(str, format, args);
	va_end(args);
	return (int)size;
}
#elif __unix__
int asprintf(char** str, const char* format, ...) {
	int size;
	va_list args;
	va_start(args, format);
	size = vasprintf(str, format, args);
	va_end(args);
	return (int)size;
}
#endif

int strtoi(char* str, int* result) {
	char* endptr;
	long val;

	errno = 0;    /* To distinguish success/failure after call */
	val = strtol(str, &endptr, 10);

	/* Check for various possible errors */

	if ((errno == ERANGE && (val == LONG_MAX || val == LONG_MIN))
		|| (errno != 0 && val == 0)) {
		return 1;
	}

	if (endptr == str) {
		errno = EINVAL;
		return 1;
	}
	/* If we got here, strtol() successfully parsed a number */

	*result = (int)val;

	return 0;
}
```

connection.h

```c
#pragma once
#ifdef _WIN32
#pragma comment (lib, "Ws2_32.lib")
#ifndef _WINSOCKAPI_
#define _WINSOCKAPI_
#endif
#include <Windows.h>
#include <WinSock2.h>
#include <WS2tcpip.h>
#endif
#include <stdint.h>

typedef void* connection_t;

#ifdef _WIN32
/**/

extern connection_t connection_init(LPCTSTR address, const uint16_t port);

/* Close connection return 1 and set properly errno on error */

extern int connection_close(const connection_t handle);

extern int connetcion_connect(const connection_t handle);

/* Get a malloc'd buffer wich contain a received message return number of byte read or -1 on
error*/
extern int connection_recv(const connection_t handle, LPTSTR* buff);

/**/
extern int connection_send(const connection_t handle, LPCTSTR buff);

#elif __unix__
/**/

extern connection_t connection_init(const char* address, const uint16_t port);

/* Close connection return 1 and set properly errno on error */

extern int connection_close(const connection_t handle);

extern int connetcion_connect(const connection_t handle);

/* Get a malloc'd buffer wich contain a received message return number of byte read or -1 on
error*/
extern int connection_recv(const connection_t handle, char** buff);

/* return number of bytes sended */

extern int connection_send(const connection_t handle, const char* buff);

/* Initiazlize connection return 1 and set properly errno on error */

extern int connection_listen(const connection_t handle);

/* Get an accepted connection return 1 and set properly errno on error */

extern connection_t connection_accepted(const connection_t handle);

#endif
```

connection.c

```
#include "connection.h"

#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

#ifdef _WIN32
	#ifndef _WINSOCKAPI_
		#define _WINSOCKAPI_
	#endif
	#include <Windows.h>
	#include <WinSock2.h>
	#include <WS2tcpip.h>
	#include <basetsd.h>
	#include <tchar.h>
#elif __unix__
	#include <unistd.h>
	#include <stddef.h>
	#include <sys/socket.h>
	#include <arpa/inet.h>
	#include <sys/types.h>
	#include <sys/un.h>
	#include <errno.h>
	#define InetPton(Family, pszAddrString, pAddrBuf) inet_aton(pszAddrString, pAddrBuf)
	#define SSIZE_T ssize_t
	#define TCHAR char
	#define LPTSTR char*
	#define LPCTSTR const char*
	#define INVALID_SOCKET -1
	#define SOCKET_ERROR  -1
	#define _tcslen strlen
	#define closesocket close
#endif

#ifdef _WIN32
struct connection {
	SOCKET socket;
	struct sockaddr* addr;
	socklen_t addrlen;
};
#elif __unix__
struct connection {
	int socket;
	struct sockaddr* addr;
	socklen_t addrlen;
};
#endif

#define BACKLOG 4096
#define MSG_LEN 4096

connection_t connection_init(LPCTSTR address, const uint16_t port) {
	struct connection* connection;
	if ((connection = malloc(sizeof(struct connection))) == NULL) {
		return NULL;
	}
#ifdef __unix__
	/*	If port is zero create a unix socket	*/
	if (!port) {
		struct sockaddr_un* paddr_un;
		if ((connection->socket = socket(AF_UNIX, SOCK_STREAM, 0)) == INVALID_SOCKET)
{
			free(connection);
			return NULL;
		}
		if ((paddr_un = malloc(sizeof(struct sockaddr_un))) == NULL) {
			free(connection);
			return NULL;
		}
		memset(paddr_un, 'x', sizeof(struct sockaddr_un));
		paddr_un->sun_family = AF_UNIX;
		paddr_un->sun_path[0] = '\0';
		strncpy(paddr_un->sun_path + 1, address, strlen(address));
		connection->addr = (struct sockaddr*)paddr_un;
		connection->addrlen = (socklen_t)(offsetof(struct sockaddr_un, sun_path) + 1 +
strlen(address));
		return connection;
	}
#endif
	struct sockaddr_in* paddr_in;
	struct in_addr haddr;		//host address
#ifdef _WIN32
	WSADATA WSAData;
	if (WSAStartup(MAKEWORD(1, 1), &WSAData) != 0) {
		free(connection);
		return NULL;
	}
#endif
	if ((connection->socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) ==
INVALID_SOCKET) {
		free(connection);
		return NULL;
	}
	if ((paddr_in = malloc(sizeof(struct sockaddr_in))) == NULL) {
		free(connection);
		return NULL;
	}
	if (!InetPton(AF_INET, address, &haddr)) {
		free(connection);
		free(paddr_in);
#ifdef __unix__
		errno = EINVAL;
#endif
		return NULL;
	}
	memset(&connection->addr, 0, sizeof(connection->addr));
	paddr_in->sin_family = AF_INET;
	paddr_in->sin_port = htons(port);
	paddr_in->sin_addr = haddr;
	connection->addr = (struct sockaddr*)paddr_in;
	connection->addrlen = sizeof(struct sockaddr_in);
	return connection;
}

int connection_close(const connection_t handle){
	struct connection* connection = (struct connection*)handle;
	if (closesocket(connection->socket) == -1){
		return -1;
	}
	free(connection->addr);
	free(connection);
#ifdef _WIN32
	WSACleanup();
```

```
#endif
	return 0;
}

int connection_recv(const connection_t handle, LPTSTR* buff) {
	struct connection* connection = (struct connection*)handle;
#ifdef _UNICODE
	char* utf8_buff;
#else
	#define utf8_buff (*buff)
#endif
	SSIZE_T len;
	if ((utf8_buff = malloc(sizeof(char) * (MSG_LEN + 1))) == NULL) {
		return -1;
	}
	memset(utf8_buff, 0, sizeof(char) * (MSG_LEN + 1));
	len = recv(connection->socket, utf8_buff, MSG_LEN, 0);
	if (len == -1 || len > MSG_LEN) {
		free(utf8_buff);
		utf8_buff = NULL;
		return -1;
	}
#ifdef _UNICODE
	int str_len;
	if (!(str_len = MultiByteToWideChar(CP_UTF8, 0, (LPCCH)utf8_buff, -1, NULL, 0))) {
		free(utf8_buff);
		return -1;
	}
	if ((*buff = malloc(sizeof(WCHAR) * str_len)) == NULL) {
		free(utf8_buff);
		return -1;
	}
	if (!MultiByteToWideChar(CP_UTF8, 0, (LPCCH)utf8_buff, -1, *buff, str_len)) {
		free(buff);
		free(utf8_buff);
		buff = NULL;
		return -1;
	}
	free(utf8_buff);
#else
	#undef utf8_buff
#endif
	return (int)len;
}

int connection_send(const connection_t handle, LPCTSTR buff) {
	struct connection* connection = (struct connection*)handle;
	int len;
#ifdef _UNICODE
	char* utf8_buff = NULL;
	if (!(len = WideCharToMultiByte(CP_UTF8, WC_ERR_INVALID_CHARS, buff, -1, NULL, 0,
NULL, NULL))) {
		return -1;
	}
	if ((utf8_buff = malloc(sizeof(char) * len)) == NULL) {
		return -1;
	}
	if (!WideCharToMultiByte(CP_UTF8, WC_ERR_INVALID_CHARS, buff, -1, utf8_buff,
(int)len, NULL, NULL)) {
		free(utf8_buff);
		return -1;
	}
	if ((len = send(connection->socket, utf8_buff, (int)strlen(utf8_buff), 0)) == -1) {
		free(utf8_buff);
```

```c
			return -1;
		}
		free(utf8_buff);
#else
		if ((len = (int)send(connection->socket, buff, strlen(buff), 0)) == -1) {
			return -1;
		}
#endif
		return len;
}

int connetcion_connect(const connection_t handle) {
		struct connection* connection = (struct connection*)handle;
		if (connect(connection->socket, connection->addr, connection->addrlen) ==
SOCKET_ERROR) {
			return -1;
		}
		return 0;
}

#ifdef __unix__

int connection_listen(const connection_t handle) {
		struct connection* connection = (struct connection*)handle;
		if (bind(connection->socket, connection->addr, connection->addrlen) == -1) {
			return -1;
		}
		if (listen(connection->socket, BACKLOG) == -1) {
			return -1;
		}
		return 0;
}

connection_t connection_accepted(const connection_t handle) {
		struct connection* listener = (struct connection*)handle;
		struct connection* accepted;
		if ((accepted = malloc(sizeof(struct connection))) == NULL) {
			return NULL;
		}
		memset(&accepted->addrlen, 0, sizeof(socklen_t));
		accepted->addr = malloc(sizeof(accepted->addrlen));
		while ((accepted->socket = accept(listener->socket, accepted->addr, &accepted-
>addrlen)) == -1) {
			if (errno != EMFILE) {
				free(accepted);
				return NULL;
			}
			sleep(1);
		}
		return accepted;
}

#endif
```

booking.h

```
#include <stdio.h>
#include <Windows.h>

//	Variabili globali:

typedef HANDLE HBOOKING;

//
//	FUNZIONE: InitializeBooking(LPCTSTR)
//
//	SCOPO: Crea e inizializza lo storage
//
//	RETURN:		l'handle dell'oggetto in caso di successo NULL in caso di errore
//
//	COMMENTI:
//
//		In questa funzione viene aperta la sessione al file di salvataggio, se
//		il file non esiste questo viene creato insieme alla sua directory 
d'appartenenza.
//
HBOOKING InitializeBooking(LPCTSTR);

//
//	FUNZIONE: SetBooking(HBOOKING, LPCTSTR)
//
//	SCOPO: Aggiorna la prenotazione
//
//	RETURN:		TRUE in caso di successo FALSE in caso di errore
//
BOOL SetBooking(HBOOKING, LPCTSTR);
//
//	FUNZIONE: GetBooking(HBOOKING)
//
//	SCOPO: Restituisce la stringa contenente il codice di prenotazione
//
//	RETURN:		la stringa in caso di successo NULL in caso di errore
//
LPTSTR GetBooking(HBOOKING);
```

booking.c

```
#include "booking.h"
#include <Windows.h>
#include <tchar.h>
#include <stdio.h>

#include "resources.h"

#define MAX_SIZE 32

// Dichiarazioni con prototipo di funzioni incluse in questo modulo di codice:

static LPTSTR GetFilename(LPCTSTR lpParam);

HBOOKING InitializeBooking(LPCTSTR lpParam) {
	HBOOKING hBooking = NULL;
	LPTSTR filename = NULL;

	if ((filename = GetFilename(lpParam)) == NULL) {
		return hBooking;
	}
	hBooking = CreateFile(
		filename,
		GENERIC_READ | GENERIC_WRITE,
		0,
		NULL,
		OPEN_ALWAYS,
		FILE_ATTRIBUTE_HIDDEN,
		NULL
	);
	if (hBooking == INVALID_HANDLE_VALUE) {
		hBooking = NULL;
	}
	free(filename);
	return hBooking;
}

BOOL SetBooking(HBOOKING hBooking, LPCTSTR lpParam) {
	DWORD dwBytesWritten = 0;
	SetFilePointer(hBooking, 0, NULL, FILE_BEGIN);
	SetEndOfFile(hBooking);
	return WriteFile(hBooking, lpParam, _tcslen(lpParam) * sizeof(TCHAR),
&dwBytesWritten, NULL);
}

LPTSTR GetBooking(HBOOKING hBooking) {
	LPTSTR lpBuffer = NULL;
	DWORD dwBytesRead = 0;
	DWORD dwBytesReadable = GetFileSize(hBooking, NULL);
	if (dwBytesReadable == INVALID_FILE_SIZE || dwBytesReadable > MAX_SIZE) {
		return NULL;
	}
	if ((lpBuffer = malloc(sizeof(TCHAR) * (dwBytesReadable + 1))) == NULL) {
		return NULL;
	}
	memset(lpBuffer, 0, sizeof(TCHAR) * (dwBytesReadable + 1));
	SetFilePointer(hBooking, 0, NULL, FILE_BEGIN);
	if (!ReadFile(hBooking, lpBuffer, dwBytesReadable, &dwBytesRead, NULL)) {
		return NULL;
	}
	return lpBuffer;
}
```

```
static LPTSTR GetFilename(LPCTSTR lpParam) {
	LPTSTR lpBuffer = NULL;
	LPTSTR lpTmp = NULL;
	DWORD len = 0;

	if (lpParam == NULL) {
		return NULL;
	}
	if (!(len = GetEnvironmentVariable(TEXT("APPDATA"), lpBuffer, 0))) {
		return NULL;
	}
	len++;
	if ((lpBuffer = malloc(sizeof(TCHAR) * len)) == NULL) {
		return NULL;
	}
	if (!GetEnvironmentVariable(TEXT("APPDATA"), lpBuffer, len)) {
		free(lpBuffer);
		return NULL;
	}
	lpTmp = lpBuffer;
	if (asprintf(&lpBuffer, TEXT("%s\\%s"), lpBuffer, lpParam) == -1) {
		free(lpBuffer);
		return NULL;
	}
	free(lpTmp);
	if (!CreateDirectory(lpBuffer, NULL) && GetLastError() != ERROR_ALREADY_EXISTS) {
		free(lpBuffer);
		return NULL;
	}
	lpTmp = lpBuffer;
	if (asprintf(&lpBuffer, TEXT("%s\\%s"), lpBuffer, TEXT("sav.dat")) == -1) {
		free(lpBuffer);
		return NULL;
	}
	free(lpTmp);
	return lpBuffer;
}
```

cinema-client.c

```
#include "connection.h"
#include "cinema-client.h"
#include "framework.h"
#include "booking.h"
#include "resources.h"
#ifdef _DEBUG
#include <io.h>
#include <stdio.h>
#include <fcntl.h>
#include <tchar.h>
#endif

#define WS_CSTYLE WS_OVERLAPPEDWINDOW ^ WS_THICKFRAME ^ WS_MAXIMIZEBOX

//     Variabili globali:
HINSTANCE hInst;           // Istanza corrente
LPTSTR szTitle;                   // Testo della barra del titolo
LPTSTR szWindowClass;      // Nome della classe della finestra principale
int rows;
int columns;
HBOOKING hBooking;
HWND hButton1;
HWND hButton2;
HWND hButton3;
HWND hStaticTextbox;
HWND* hStaticS;      //Pointer to seat control vector
HWND hStaticLabelScreen;
HBITMAP hBitmapDefault;
HBITMAP hBitmapBooked;
HBITMAP hBitmapSelected;
HBITMAP hBitmapRemove;
HBITMAP hBitmapDisabled;

// Dichiarazioni con prototipo di funzioni incluse in questo modulo di codice:

ATOM                MyRegisterClass(HINSTANCE);
BOOL                InitInstance(HINSTANCE, int);
LRESULT CALLBACK    WndProc(HWND, UINT, WPARAM, LPARAM);
INT_PTR CALLBACK    About(HWND, UINT, WPARAM, LPARAM);
BOOL                       ButtonClickHandler(HWND, LPCTSTR*);
BOOL                       UpdateSeats(HWND, BOOL);
BOOL                       QueryServer(LPCTSTR, LPTSTR*);
BOOL                       GetSeatsQuery(LPTSTR*, HBITMAP);
void                       ErrorHandler(int e);

int APIENTRY WinMain(_In_ HINSTANCE hInstance, _In_opt_ HINSTANCE hPrevInstance, _In_ LPTSTR
lpCmdLine, _In_ int nCmdShow) {

       UNREFERENCED_PARAMETER(hPrevInstance);
       UNREFERENCED_PARAMETER(lpCmdLine);

#ifdef _DEBUG
       int hCrt;

       AllocConsole();

       HANDLE handle_in = GetStdHandle(STD_INPUT_HANDLE);
       hCrt = _open_osfhandle((long)handle_in, _O_TEXT);
       FILE* hf_in = _fdopen(hCrt, "r");
       _tfreopen_s(&hf_in, TEXT("CONIN$"), TEXT("r"), stdin);

       HANDLE handle_out = GetStdHandle(STD_OUTPUT_HANDLE);
```

```
	hCrt = _open_osfhandle((long)handle_out, _O_TEXT);
	FILE* hf_out = _fdopen(hCrt, "w");
	_tfreopen_s(&hf_out, TEXT("CONOUT$"), TEXT("w"), stdout);

	HANDLE handle_err = GetStdHandle(STD_ERROR_HANDLE);
	hCrt = _open_osfhandle((long)handle_err, _O_TEXT);
	FILE* hf_err = _fdopen(hCrt, "w");
	_tfreopen_s(&hf_err, TEXT("CONOUT$"), TEXT("w"), stderr);
#endif

	//	Inizializzare le variabili globali
	LPTSTR buffer;

	szTitle = TEXT("Prenotazione");
	szWindowClass = TEXT("generic_class");

	//	Create the booking
	hBooking = InitializeBooking(TEXT("PrenotazioneCinema"));
#ifdef _DEBUG
	_tprintf(TEXT("BOOKING CREATED\n"));
#endif

	if (hBooking == NULL) {
		ErrorHandler(GetLastError());
	}
	//	Retrive number of seats and rows
	if (!QueryServer(TEXT("GET ROWS"), &buffer)) {
		ErrorHandler(WSAGetLastError());
	}
	rows = _tstoi(buffer);
	free(buffer);
	if (!QueryServer(TEXT("GET COLUMNS"), &buffer)) {
		ErrorHandler(WSAGetLastError());
	}
	columns = _tstoi(buffer);
	free(buffer);
	//	Initialize seats buttons
	hStaticS = malloc((size_t)(rows * columns) * sizeof(HWND));

	if (!MyRegisterClass(hInstance)) {
		ErrorHandler(GetLastError());
	}

#ifdef _DEBUG
	_tprintf(TEXT("WINDOW CLASS REGITRATED\n"));
#endif

	// Eseguire l'inizializzazione dall'applicazione:
	if (!InitInstance(hInstance, nCmdShow)) {
		ErrorHandler(GetLastError());
	}

#ifdef _DEBUG
	_tprintf(TEXT("WINDOW INITIALIZATED\n"));
#endif
	MSG msg;

	// Ciclo di messaggi principale:
	while (GetMessage(&msg, NULL, 0, 0) > 0) {
		TranslateMessage(&msg);
		DispatchMessage(&msg);
	}

#ifdef _DEBUG
```

```c
	_tprintf(TEXT("RECEIVED DESTROY OR INVALID MESSAGE\n"));
#endif

	CloseHandle(hBooking);
	free(hStaticS);
	return (int)msg.wParam;
}

//
//  FUNZIONE: MyRegisterClass()
//
//  SCOPO: Registra la classe di finestre.
//
ATOM MyRegisterClass(HINSTANCE hInstance) {

	WNDCLASS wndc;

	wndc.style = CS_HREDRAW | CS_VREDRAW;
	wndc.lpfnWndProc = WndProc;
	wndc.cbClsExtra = 0;
	wndc.cbWndExtra = 0;
	wndc.hInstance = hInstance;
	wndc.hIcon = LoadIcon(hInstance, IDI_APPLICATION);
	wndc.hCursor = LoadCursor(NULL, IDC_ARROW);
	wndc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
	wndc.lpszMenuName = NULL;
	wndc.lpszClassName = (LPCTSTR)szWindowClass;

	return RegisterClass(&wndc);
}

//
//   FUNZIONE: InitInstance(HINSTANCE, int)
//
//   SCOPO: Salva l'handle di istanza e crea la finestra principale
//
//   COMMENTI:
//
//        In questa funzione l'handle di istanza viene salvato in una variabile globale e
//        viene creata e visualizzata la finestra principale del programma.
//
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow) {
	HWND hWnd;
	int XRes = 1280;
	int YRes = 720;

	hInst = hInstance;  //     Archivia l'handle di istanza nella variabile globale

	if (!(hWnd = CreateWindow(
		(LPCTSTR)szWindowClass,
		//     CLASS
		(LPCTSTR)szTitle,
		//     TITLE
		WS_OVERLAPPEDWINDOW ^ WS_THICKFRAME ^ WS_MAXIMIZEBOX,        //     STYLE
		CW_USEDEFAULT,
			//     X
		CW_USEDEFAULT,
			//     Y
		XRes,
			//     WIDTH
		YRes,
			//     HEIGHT
		NULL,
			//     NO PARENT WINDOW
		NULL,
			//	NO MENU
		hInstance,
			//	INSTANCE
		NULL
			//	NO PARAMETER
	))) return FALSE;
#ifdef _DEBUG
	_tprintf(TEXT("WINDOW SUCCESFULLY CREATED\n"));
#endif

	//	Inizializza le risorse
	{
		if ((hBitmapDisabled = LoadBitmap(hInstance, MAKEINTRESOURCE(IDB_BITMAP1))) ==
NULL) {
			ErrorHandler(GetLastError());
		}
		if ((hBitmapDefault = LoadBitmap(hInstance, MAKEINTRESOURCE(IDB_BITMAP2))) ==
NULL) {
			ErrorHandler(GetLastError());
		}
		if ((hBitmapBooked = LoadBitmap(hInstance, MAKEINTRESOURCE(IDB_BITMAP3))) ==
NULL) {
			ErrorHandler(GetLastError());
		}
		if ((hBitmapSelected = LoadBitmap(hInstance, MAKEINTRESOURCE(IDB_BITMAP4))) ==
NULL) {
			ErrorHandler(GetLastError());
		}
		if ((hBitmapRemove = LoadBitmap(hInstance, MAKEINTRESOURCE(IDB_BITMAP5))) ==
NULL) {
			ErrorHandler(GetLastError());
		}
	}


	//	Creazione Label
	{
		LPTSTR film;
		LPTSTR showtime;
		LPTSTR buffer;
		if (!(CreateWindow(
			TEXT("STATIC"),					//
	PREDEFINED CLASS
			TEXT("Codice prenotazione:"),			//	text
			WS_CHILD | WS_VISIBLE,				//	Styles
			(XRes / 2) - (210 / 2) - 170,			//	x position
			15,						//
	y position
			140, 20,					//	w,h
size
			hWnd, NULL, hInstance,				//	PARENT
WINDOW, MENU, INSTANCE
			NULL						//
	PARAMETER
		))) return FALSE;

		if (!QueryServer(TEXT("GET FILM"), &buffer)) {
			ErrorHandler(WSAGetLastError());
		}
		if (asprintf(&film, TEXT("Film: %s"), buffer) == -1) {
			return FALSE;
		}
		free(buffer);
```

```
              if (!(CreateWindow(
                    TEXT("STATIC"),                                               //
       PREDEFINED CLASS
                    film,                                                         //    text
                    WS_CHILD | WS_VISIBLE,                                 //     Styles
                    10, 5,                                                        //    x,y
position
                    255, 20,                                                      //    w,h
size
                    hWnd, NULL, hInstance,                                 //     PARENT
WINDOW, MENU, INSTANCE
                    NULL                                                          //
       PARAMETER
              ))) return FALSE;
              free(film);

              if (!QueryServer(TEXT("GET SHOWTIME"), &buffer)) {
                    ErrorHandler(WSAGetLastError());
              }
              if (asprintf(&showtime, TEXT("Orario: %s"), buffer) == -1) {
                    return FALSE;
              }
              free(buffer);

              if (!(CreateWindow(
                    TEXT("STATIC"),                                               //
       PREDEFINED CLASS
                    showtime,                                                     //    text
                    WS_CHILD | WS_VISIBLE,                                 //     Styles
                    10, 25,                                                             //
       x,y position
                    255, 20,                                                      //    w,h
size
                    hWnd, NULL, hInstance,                                 //     PARENT
WINDOW, MENU, INSTANCE
                    NULL                                                          //
       PARAMETER
              ))) return FALSE;
              free(showtime);

              if (!(hStaticLabelScreen = CreateWindow(
                    TEXT("STATIC"),                                               //
       PREDEFINED CLASS
                    TEXT("SCHERMO"),                                       //     text
                    WS_CHILD | WS_VISIBLE | SS_CENTER,              //     Styles
                    (XRes / 2) - (16 * columns) - 24,        //     x position
                    48 + ((YRes - 150) / 2) + (16 * rows),   //     y position
                    (32 * columns) + 48, 20,                        //     w,h size
                    hWnd, NULL, hInstance,                                 //     PARENT
WINDOW, MENU, INSTANCE
                    NULL                                                          //
       PARAMETER
              ))) return FALSE;

       }

       //     Creazione Textbox
       {
              LPTSTR bookingCode;

              if ((bookingCode = GetBooking(hBooking)) == NULL) {
                    ErrorHandler(GetLastError());
              }
```

```
        if (!(hStaticTextbox = CreateWindowEx(
            WS_EX_CLIENTEDGE,                       //  EX style
            TEXT("STATIC"),                             //  PREDEFINED
CLASS
            bookingCode,                            //  text
            WS_CHILD | WS_VISIBLE | SS_CENTER,  //  Styles
            (XRes / 2) - (210 / 2),                 //  x position
            15,                                             //  y
position
            210, 20,                                    //  w,h size
            hWnd, NULL, hInstance,                  //  PARENT WINDOW,
MENU, INSTANCE
            NULL                                        //  PARAMETER
        ))) return FALSE;
        free(bookingCode);
#ifdef _DEBUG
        _tprintf(TEXT("TEXTBOX CREATED\n"));
#endif
    }

    //  Create Button
    {
        if (!(hButton1 = CreateWindow(
            TEXT("BUTTON"),                         //  PREDEFINED CLASS
            TEXT("Prenota"),                    //  Button text
            WS_CHILD,                               //  Styles
            (XRes / 2) - (210 / 2),             //  x position
            YRes - (2 * 60),                    //  y position
            210, 60,                                //  w,h size
            hWnd, NULL, hInstance,              //  PARENT WINDOW, MENU,
INSTANCE
            NULL                                    //  PARAMETER
        ))) return FALSE;

        //  Create Button
        if (!(hButton2 = CreateWindow(
            TEXT("BUTTON"),                         //  PREDEFINED CLASS
            TEXT("Modifica prenotazione"),  //  Button text
            WS_CHILD,                               //  Styles
            (XRes / 2) - (210 * 4 / 3),     //  x position
            YRes - (2 * 60),                    //  y position
            210, 60,                                //  w,h SIZE
            hWnd, NULL, hInstance,              //  PARENT WINDOW, MENU,
INSTANCE
            NULL                                    //  PARAMETER
        ))) return FALSE;

        //  Create Button
        hButton3 = CreateWindow(
            TEXT("BUTTON"),                         //  PREDEFINED CLASS
            TEXT("Elimina prenotazione"),   //  Button text
            WS_CHILD,                               //  Styles
            (XRes / 2) + (210 * 2 / 7),     //  x position
            YRes - (2 * 60),                    //  y position
            210, 60,                                //  w,h size
            hWnd, NULL, hInstance,              //  PARENT WINDOW, MENU,
INSTANCE
            NULL                                    //  PARAMETER
        );
        if (!hButton3) {
            return FALSE;
        }
#ifdef _DEBUG
        _tprintf(TEXT("BUTTONS CREATED\n"));
```

```
#endif
	}

	//	Creazione Contolli Poltrone
	{

		for (int i = 0; i < rows; i++) {
			LPTSTR str;
			asprintf(&str, TEXT("%c"), i + 65);
			CreateWindow(
				TEXT("STATIC"),
				//	PREDEFINED CLASS
				str,
				//	text
				WS_CHILD | WS_VISIBLE | SS_CENTER | SS_CENTERIMAGE,		//
	Styles
				(XRes / 2) - (16 * columns) - 32,
	//	x position
				32 + ((YRes - 150) / 2) - (16 * rows) + (32 * i),		//
	y position
				32, 32,
					//	w,h size
				hWnd, NULL, hInstance,
			//	PARENT WINDOW, MENU, INSTANCE
				NULL
				//	PARAMETER
			);
			free(str);
		}

		for (int j = 0; j < columns; j++) {
			LPTSTR str;
			asprintf(&str, TEXT("%d"), j + 1);
			CreateWindow(
				TEXT("STATIC"),
				//	PREDEFINED CLASS
				str,
				//	text
				WS_CHILD | WS_VISIBLE | SS_CENTER | SS_CENTERIMAGE,		//
	Styles
				(XRes / 2) - (16 * columns) + (32 * j),
	//	x position
				((YRes - 150) / 2) - (16 * rows),
	//	y position
				32, 32,
					//	w,h size
				hWnd, NULL, hInstance,
			//	PARENT WINDOW, MENU, INSTANCE
				NULL
				//	PARAMETER
			);
			free(str);
		}

		for (int i = 0; i < rows; i++) {
			for (int j = 0; j < columns; j++) {
				hStaticS[(i * columns) + j] = CreateWindow(
					TEXT("STATIC"),
				//	PREDEFINED CLASS
					TEXT(""),
				//	text
					WS_CHILD | WS_VISIBLE | SS_BITMAP | SS_NOTIFY,		//
	Styles
```

```
                                        (XRes / 2) - (16 * columns) + (32 * j),
            //      x position
                                        32 + ((YRes - 150) / 2) - (16 * rows) + (32 * i),      //
            y position
                                        32, 32,
                                        //      w,h size
                                        hWnd, NULL, hInstance,
                        //      PARENT WINDOW, MENU, INSTANCE
                                        NULL
                                //      PARAMETER
                                );
                                if (!hStaticS[(i * columns) + j]) {
                                        return FALSE;
                                }
                                SendMessage(hStaticS[(i * columns) + j], STM_SETIMAGE,
(WPARAM)IMAGE_BITMAP, (LPARAM)hBitmapDefault);
                        }
                }

                if (!UpdateSeats(hWnd, FALSE)) {
                        ErrorHandler(GetLastError());
                }
        }

        ShowWindow(hWnd, nCmdShow);
        UpdateWindow(hWnd);
        return TRUE;
}

//
//  FUNZIONE: WndProc(HWND, UINT, WPARAM, LPARAM)
//
//  SCOPO: Elabora i messaggi per la finestra principale.
//
//  WM_COMMAND  - elabora il menu dell'applicazione
//  WM_PAINT    - Disegna la finestra principale
//  WM_DESTROY  - inserisce un messaggio di uscita e restituisce un risultato
//
//
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam) {
        switch (message) {
        case WM_CREATE: {
#ifdef _DEBUG
                _tprintf(TEXT("RECEIVED CREATE MESSAGE\n"));
#endif
                if (!SetTimer(hWnd, 0, 1000, (TIMERPROC)NULL)) {
                        ErrorHandler(GetLastError());
                }
        }
                break;
        case WM_PAINT: {
#ifdef _DEBUG
                _tprintf(TEXT("RECEIVED PAINT MESSAGE\n"));
#endif
                LPTSTR bookingCode;
                PAINTSTRUCT ps;
                HDC hdc;

                hdc = BeginPaint(hWnd, &ps);
                //      TODO: Aggiungere qui il codice di disegno che usa HDC...
                EndPaint(hWnd, &ps);

                if ((bookingCode = GetBooking(hBooking)) == NULL) {
                        ErrorHandler(GetLastError());
```

```
			}
			if (!_tcscmp(bookingCode, TEXT(""))) {
				ShowWindow(hButton1, SW_SHOWNORMAL);
				ShowWindow(hButton2, SW_HIDE);
				ShowWindow(hButton3, SW_HIDE);
			}
			else {
				ShowWindow(hButton1, SW_HIDE);
				ShowWindow(hButton2, SW_SHOWNORMAL);
				ShowWindow(hButton3, SW_SHOWNORMAL);
			}
			free(bookingCode);
		}
			return DefWindowProc(hWnd, message, wParam, lParam);
		case WM_CTLCOLORSTATIC: {
			if ((HWND)lParam == hStaticLabelScreen) {
				SetBkColor((HDC)wParam, GetSysColor(COLOR_SCROLLBAR));
				return (LRESULT)GetSysColorBrush(COLOR_SCROLLBAR);
			}
			else if ((HWND)lParam != hStaticTextbox) {
				return (LRESULT)GetSysColorBrush(COLOR_WINDOW);
			}
		}
			break;
		case WM_TIMER: {
#ifdef _DEBUG
			_tprintf(TEXT("RECEIVED TIMER MESSAGE\n"));
#endif
			if (!UpdateSeats(hWnd, FALSE)) {
				ErrorHandler(GetLastError());
			}
		}
			break;
		case WM_COMMAND: {
			if (wParam == BN_CLICKED) {
#ifdef _DEBUG
				_tprintf(TEXT("RECEIVED WM_COMMAND CLICKED MESSAGE FROM CONTROL %d\n"),
lParam);
#endif

				/*	Static Control		*/
				{
					HBITMAP tmp;

					tmp = (HBITMAP)SendMessage((HWND)lParam, STM_GETIMAGE,
(WPARAM)IMAGE_BITMAP, 0);
					if (tmp != NULL) {
						if (tmp == hBitmapDefault) {
							SendMessage((HWND)lParam, STM_SETIMAGE,
(WPARAM)IMAGE_BITMAP, (LPARAM)hBitmapSelected);
						}
						else if (tmp == hBitmapSelected) {
							SendMessage((HWND)lParam, STM_SETIMAGE,
(WPARAM)IMAGE_BITMAP, (LPARAM)hBitmapDefault);
						}
						else if (tmp == hBitmapBooked) {
							SendMessage((HWND)lParam, STM_SETIMAGE,
(WPARAM)IMAGE_BITMAP, (LPARAM)hBitmapRemove);
						}
						else if (tmp == hBitmapRemove) {
							SendMessage((HWND)lParam, STM_SETIMAGE,
(WPARAM)IMAGE_BITMAP, (LPARAM)hBitmapBooked);
						}
						else {
```

```c
                        return DefWindowProc(hWnd, message, wParam,
lParam);
                    }
                    return 0;
                }
            }
            /*     Button Control      */
            LPTSTR bookingCode = NULL;
            LPTSTR* queries = NULL;

            if ((bookingCode = GetBooking(hBooking)) == NULL) {
                ErrorHandler(GetLastError());
            }

            if ((HWND)lParam == hButton1) {
                if ((queries = malloc(sizeof(LPTSTR) * 2)) == NULL) {
                    ErrorHandler(GetLastError());
                }
                if (asprintf(&queries[0], TEXT("BOOK -1")) == -1) {
                    ErrorHandler(GetLastError());
                }
                if (!GetSeatsQuery(&queries[0], hBitmapSelected)) {
                    free(queries[0]);
                    free(queries);
                    free(bookingCode);
                    return 0;
                }
                queries[1] = NULL;
                if (!ButtonClickHandler(hWnd, queries)) {
                    ErrorHandler(GetLastError());
                }
                free(queries[0]);
                free(queries);
            }
            else if ((HWND)lParam == hButton2) {
                int i = 1;

                if ((queries = malloc(sizeof(LPTSTR) * 3)) == NULL) {
                    ErrorHandler(GetLastError());
                }
                if (asprintf(&queries[0], TEXT("BOOK %s"), bookingCode) == -1) {
                    ErrorHandler(GetLastError());
                }
                if (!GetSeatsQuery(&queries[0], hBitmapSelected)) {
                    free(queries[0]);
                    queries[0] = NULL;
                    queries[1] = NULL;
                    i = 0;
                }
                if (asprintf(&queries[i], TEXT("DELETE %s"), bookingCode) == -1)
{

                    ErrorHandler(GetLastError());
                }
                if (!GetSeatsQuery(&queries[i], hBitmapRemove)) {
                    free(queries[i]);
                    queries[i] = NULL;
                }
                queries[2] = NULL;
                if (!ButtonClickHandler(hWnd, queries)) {
                    ErrorHandler(GetLastError());
                }
                free(queries[0]);
                if (i) {
                    free(queries[1]);
```

```
				}
				free(queries);
			}
			else if ((HWND)lParam == hButton3) {
				int MessageBoxResult;

				MessageBoxResult = MessageBox(
					hWnd,
					TEXT("Sei sicuro di voler eliminare la tua
prenotazione?"),
					TEXT("Elimina prenotazione"),
					MB_YESNO | MB_ICONWARNING | MB_DEFBUTTON2 | MB_APPLMODAL
				);
				if (MessageBoxResult == IDYES) {
					BOOL booking;
					if ((queries = malloc(sizeof(LPTSTR) * 2)) == NULL) {
						ErrorHandler(GetLastError());
					}
					if (asprintf(&queries[0], TEXT("DELETE %s"), bookingCode)
== -1) {

						ErrorHandler(GetLastError());
					}
					booking = GetSeatsQuery(&queries[0], hBitmapBooked);
					booking |= GetSeatsQuery(&queries[0], hBitmapRemove);
					if (!booking) {
						free(queries[0]);
						free(queries);
						free(bookingCode);
						return 0;
					}
					queries[1] = NULL;
					if (!ButtonClickHandler(hWnd, queries)) {
						ErrorHandler(GetLastError());
					}
					free(queries[0]);
					free(queries);
				}
			}

			free(bookingCode);
		}
		return DefWindowProc(hWnd, message, wParam, lParam);
	}
	case WM_DRAWITEM: {
		(LPDRAWITEMSTRUCT)lParam;
	}
		return DefWindowProc(hWnd, message, wParam, lParam);
	case WM_DESTROY: {
#ifdef _DEBUG
		_tprintf(TEXT("RECEIVED DESTROY MESSAGE\n"));
#endif
		PostQuitMessage(0);
	}
		break;
	default:
		return DefWindowProc(hWnd, message, wParam, lParam);
	}
	return 0;
}

BOOL UpdateSeats(HWND hWnd, BOOL reset) {
	LPTSTR query;
	LPTSTR result;
	LPTSTR bookingCode;
	BOOL booking = FALSE;

	if ((bookingCode = GetBooking(hBooking)) == NULL) {
		return FALSE;
	}
	if (!strcmp(bookingCode, TEXT(""))) {
		if (asprintf(&query, TEXT("MAP -1")) == -1) {
			return FALSE;
		}
	}
	else {
		if (asprintf(&query, TEXT("MAP %s"), bookingCode) == -1) {
			return FALSE;
		}
	}
	free(bookingCode);
	if (!QueryServer(query, &result)) {
		ErrorHandler(WSAGetLastError());
	}
	free(query);
	for (int i = 0; i < rows * columns; i++) {
		if (result[i * 2] == TEXT('1')) {
			booking = TRUE;
		}
		if (reset) {
			if (result[i * 2] == TEXT('0')) {
				SendMessage(hStaticS[i], STM_SETIMAGE, (WPARAM)IMAGE_BITMAP,
(LPARAM)hBitmapDefault);
			}
			else if (result[i * 2] == TEXT('1')) {
				SendMessage(hStaticS[i], STM_SETIMAGE, (WPARAM)IMAGE_BITMAP,
(LPARAM)hBitmapBooked);
			}
			else {
				SendMessage(hStaticS[i], STM_SETIMAGE, (WPARAM)IMAGE_BITMAP,
(LPARAM)hBitmapDisabled);
			}
		}
		else {
			HBITMAP tmp;
			if ((tmp = (HBITMAP)SendMessage(hStaticS[i], STM_GETIMAGE,
(WPARAM)IMAGE_BITMAP, 0)) == NULL) {
				return FALSE;
			}
			if (tmp == hBitmapDefault) {
				if (result[i * 2] == TEXT('1')) {
					SendMessage(hStaticS[i], STM_SETIMAGE,
(WPARAM)IMAGE_BITMAP, (LPARAM)hBitmapBooked);
				}
				else if (result[i * 2] == TEXT('2')) {
					SendMessage(hStaticS[i], STM_SETIMAGE,
(WPARAM)IMAGE_BITMAP, (LPARAM)hBitmapDisabled);
				}
			}
			else if (((tmp == hBitmapDisabled) || (tmp == hBitmapBooked)) &&
result[i * 2] == TEXT('0')) {
				SendMessage(hStaticS[i], STM_SETIMAGE, (WPARAM)IMAGE_BITMAP,
(LPARAM)hBitmapDefault);
			}
		}
	}
	free(result);
	if (!booking) {
		if (!SetBooking(hBooking, TEXT(""))) {
				ErrorHandler(GetLastError());
			}
			SendMessage(hStaticTextbox, WM_SETTEXT, _tcslen(TEXT("")), (LPARAM)TEXT(""));
		}
		RedrawWindow(hWnd, NULL, NULL, RDW_INVALIDATE | RDW_UPDATENOW);
		return TRUE;
}

BOOL GetSeatsQuery(LPTSTR* lppQuery, HBITMAP hBitmapType) {
		BOOL result = FALSE;
		HBITMAP hBitmap = NULL;
		LPTSTR lpTmp = NULL;

		for (int i = 0; i < rows * columns; i++) {
			hBitmap = (HBITMAP)SendMessage(hStaticS[i], STM_GETIMAGE,
(WPARAM)IMAGE_BITMAP, 0);
			if (hBitmap == hBitmapType) {
				result = TRUE;
				lpTmp = *lppQuery;
				if (asprintf(lppQuery, TEXT("%s %d"), *lppQuery, i) == -1) {
					ErrorHandler(GetLastError());
				}
				free(lpTmp);
			}
		}
		return result;

}

BOOL ButtonClickHandler(HWND hWnd, LPCTSTR* queries) {
		LPTSTR buffer;

		for (int i = 0; queries[i] != NULL; i++) {
			if (!QueryServer(queries[i], &buffer)) {
				ErrorHandler(WSAGetLastError());
			}
			if (!(_tcscmp(buffer, TEXT("OPERATION FAILED")))) {
				MessageBox(
					hWnd,
					TEXT("Prenotazione falita, in caso di prenotazioni simultanee una
prenotazione potrebbe fallire.\nRiprovare"),
					NULL,
					MB_OK | MB_ICONERROR | MB_APPLMODAL
				);
			}
			if ((_tcscmp(buffer, TEXT("OPERATION SUCCEDED")))) {
				if (!SetBooking(hBooking, buffer)) {
					ErrorHandler(GetLastError());
				}
				SendMessage(hStaticTextbox, WM_SETTEXT, _tcslen(buffer),
(LPARAM)buffer);
			}
			free(buffer);
		}
		if (!UpdateSeats(hWnd, TRUE)) {
			ErrorHandler(GetLastError());
		}
		return TRUE;
}

BOOL QueryServer(LPCTSTR query, LPTSTR* result) {
		connection_t connection;
		if ((connection = connection_init(TEXT("127.0.0.1"), 55555)) == NULL) {
			return FALSE;
```

```
        }
        if (connetcion_connect(connection) == -1) {
                return FALSE;
        }
        if (connection_send(connection, query) == -1) {
                return FALSE;
        }
        while (connection_recv(connection, result) == -1) {
                return FALSE;
        }
#ifdef _DEBUG
        _tprintf(TEXT("QUERY: %s\nRESULT: %s\n"), query, *result);
#endif
        if (connection_close(connection) == -1) {
                return FALSE;
        }
        return TRUE;
}

void ErrorHandler(int e) {
        LPTSTR p_errmsg = NULL;
        FormatMessage(
                FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM |
FORMAT_MESSAGE_IGNORE_INSERTS,
                NULL,
                e,
                MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
                (LPTSTR)&p_errmsg,
                0,
                NULL
        );
#ifdef _DEBUG
        _ftprintf(stderr, TEXT("%s\n"), p_errmsg);
#endif
        FatalAppExit(
                0,
                p_errmsg
        );
}
```

cinemactl.c

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
#include <time.h>

#include "connection.h"
#include "resources.h"

#define try(foo, err_value)\
	if ((foo) == (err_value)){\
		fprintf(stderr, "%m\n");\
		exit(EXIT_FAILURE);\
	}

int server_start();
int server_stop();
int server_status();
int server_query(char*, char**);

int main(int argc, char *argv[]){
	if (argc == 2 && !strncasecmp(argv[1], "start", 5)) {
try(
		server_start(), (1)
)
	}
	else if (argc == 2 && !strncasecmp(argv[1], "stop", 4)) {
try(
		server_stop(), (1)
)
	}
	else if (argc == 2 && !strncasecmp(argv[1], "status", 6)) {
try(
		server_status(), (1)
)
	}
	else if (argc == 3 && !strncasecmp(argv[1], "query", 5)) {
		char* buff;
try(
		server_query(argv[2], &buff), (1)
)
		printf("%s\n", buff);
	}
	else {
		printf("\nUsage:\n cinemactl [COMMAND]\n\n\
			\rCommands:\n\
			\r start\n\
			\r stop\n\
			\r status\n\
			\r restart\n\
			\r query [...]\n\n");
		exit(EXIT_SUCCESS);
	}
	return 0;
}

int server_start(){
	pid_t pid;
	char *filename;
```

```c
	if (asprintf(&filename, "%s%s", getenv("HOME"), "/.cinema/bin/cinemad") == -1) {
		return 1;
	}
	pid = fork();
switch (pid) {
case 0:
	if (execl(filename, "cinemad", NULL) == -1) {
		free(filename);
		return 1;
	}
case -1:
	free(filename);
	return 1;
default:
	free(filename);
	return 0;
	}

}

int server_stop(){
	char* pid;
try(
	server_query("GET PID", &pid), (1)
)
	int pid_value;
try(
	strtoi(pid, &pid_value), (1)
)
try(
	kill(pid_value, SIGTERM), (-1)
)
	free(pid);
	return 0;
}

int server_status() {
	char* pid = NULL;
	char* timestr = NULL;
	char* icon = NULL;
	char* status = NULL;
	if (server_query("GET PID", &pid) == 1 ||
		server_query("GET TIMESTAMP", &timestr) == 1)   {
		if (asprintf(&icon, "●") == -1) {
			return 1;
		}
		if (asprintf(&status, "inactive (dead)") == -1) {
			return 1;
		}
	}
	else {
		time_t rawtime;
		struct tm* timeinfo;
		rawtime = atoll(timestr);
		timeinfo = localtime(&rawtime);
		free(timestr);
		if ((timestr = malloc(sizeof(char) * 64)) == NULL) {
			return 1;
		}
		if (strftime(timestr, 64, "%a %F %T %Z", timeinfo) == -1) {
			return 1;
		}
		if (asprintf(&icon, "\e[0;92m●\e[0m") == -1) {
			return 1;
		}
		if (asprintf(&status, "\e[1;92mactive (running)\e[0m since %s", timestr) == -
1) {
			return 1;
		}
		free(timestr);
	}
	printf("%s cinemad - The Reservation Cinema Server\n   Active: %s\n", icon, status);
	if (pid) {
		printf("Main PID : %s (cinemad)\n", pid);
		free(pid);
	}
	free(icon);
	free(status);
	return 0;
}

int server_query(char* query, char** result) {
	connection_t connection;
	char* filename;
	if (asprintf(&filename, "%s%s", getenv("HOME"), "/.cinema/tmp/socket") == -1) {
		return 1;
	}
	if ((connection = connection_init(filename, 0)) == NULL) {
		return 1;
	}
	free(filename);
	if (connetcion_connect(connection) == -1) {
		return 1;
	}
	if (connection_send(connection, query) == -1) {
		return 1;
	}
	if (connection_recv(connection, result) == -1) {
		return 1;
	}
	if (connection_close(connection) == -1) {
		return 1;
	}
	return 0;
}
```

cinemad.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <syslog.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>
#include <pthread.h>
#include <signal.h>
#include <errno.h>

#include "resources.h"
#include "database.h"
#include "connection.h"
#include "concurrent_queue.h"
#include "concurrent_flag.h"

#define try(foo, err_value)\
	if ((foo) == (err_value)){\
		syslog(LOG_ERR, "%m was generated from statment %s", #foo);\
		exit(EXIT_FAILURE);\
	}

#define TIMEOUT 5

struct request_info {
	pthread_t tid;
	connection_t connection;
};

/*	Global variables	*/

database_t database;
concurrent_queue_t request_queue;

/*	Prototype declarations of functions included in this code module	*/

void thread_exit(int sig) { pthread_exit(NULL); }	//SIAGALRM handler
void* thread_joiner(void* arg);
void* thread_timer(void* arg);
void* connection_mngr(void* arg);
void* request_handler(void* arg);
static int daemonize();

int main(int argc, char *argv[]){
	pthread_t joiner_tid;
	pthread_t internet_mngr_tid;
	pthread_t internal_mngr_tid;
	connection_t internet_connection;
	connection_t internal_connection;
	concurrent_flag_t server_status_flag;
	/*	Ignore all signals	*/
	sigset_t sigset;
try(
	sigfillset(&sigset), (-1)
)
try(
	pthread_sigmask(SIG_BLOCK, &sigset, NULL), (!0)
```

```c
)
	/*	Daemonize	*/
try(
	daemonize(), (1)
)
#ifdef _DEBUG
	syslog(LOG_DEBUG, "Main thread:\tDemonized");
#endif
	/*	Initialize status flag, request queue and start joiner thread	*/
try(
	request_queue = concurrent_queue_init(), (NULL)
)
try(
	server_status_flag = concurrent_flag_init(), (NULL)
)
try(
	concurrent_flag_set(server_status_flag), (1)
)
try(
	pthread_create(&joiner_tid, NULL, thread_joiner, server_status_flag), (!0)
)
#ifdef _DEBUG
	syslog(LOG_DEBUG, "Main thread:\tJoiner thread started");
#endif
	/*	Create directory tree	*/
try(
	mkdir("etc", 0775), (-1 * (errno != EEXIST))
)
try(
	mkdir("tmp", 0775), (-1 * (errno != EEXIST))
)
	/*	Start database	*/
	char* result;
try(
	database = database_init("etc/data.dat"), (NULL || (errno == ENOENT))
)
	if (!database) {
		int fd;
try(
		fd = open("etc/data.dat", O_RDWR | O_CREAT | O_EXCL, 0666), (-1)
)
try(
		database = database_init("etc/data.dat"), (NULL)
)
try(
		database_execute(database, "POPULATE", &result), (1)
)
		free(result);
	}
try(
	database_execute(database, "SETUP", &result), (1)
)
	free(result);
#ifdef _DEBUG
	syslog(LOG_DEBUG, "Main thread:\tDatabase connected");
#endif
	/*	Register timestamp and PID in database	*/
	char* qpid;
	char* qtsp;
try(
	asprintf(&qpid, "%s %d", "SET PID", getpid()), (-1)
)
try(
	asprintf(&qtsp, "%s %llu", "SET TIMESTAMP", (long long)time(NULL)), (-1)
```

```c
)
try(
	database_execute(database, qpid, &result), (1)
)
#ifdef _DEBUG
	syslog(LOG_DEBUG, "Main thread:\tPID stored: %s", result);
#endif
	free(result);
try(
	database_execute(database, qtsp, &result), (1)
)
#ifdef _DEBUG
	syslog(LOG_DEBUG, "Main thread:\tTIMESTAMP stored: %s", result);
#endif
	free(result);
	free(qpid);
	free(qtsp);
	/*	Setup connections	*/
	char* address;
	char* port;
try(
	database_execute(database, "GET IP", &address), (1)
)
try(
	database_execute(database, "GET PORT", &port), (1)
)
	int port_value;
try(
	strtoi(port, &port_value), (1)
)
try(
	internet_connection = connection_init(address, (uint16_t)port_value), (NULL)
)
	free(address);
	free(port);
try(
	asprintf(&address, "%s%s", getenv("HOME"), "/.cinema/tmp/socket"), (-1)
)
try(
	internal_connection = connection_init(address, 0), (NULL)
)
	free(address);
	/*	Start connection manager threads	*/
try(
	pthread_create(&internet_mngr_tid, NULL, connection_mngr, internet_connection), (!0)
)
try(
	pthread_create(&internal_mngr_tid, NULL, connection_mngr, internal_connection), (!0)
)
#ifdef _DEBUG
	syslog(LOG_DEBUG, "Main thread:\tConnection manager threads started");
#endif
	syslog(LOG_INFO, "Service started");
	/*	Wait for SIGTERM becomes pending	*/
	int sig;
try(
	sigemptyset(&sigset), (-1)
)
try(
	sigaddset(&sigset, SIGTERM), (-1)
)
#ifdef _DEBUG
	syslog(LOG_DEBUG, "Main thread:\tWait for SIGTERM");
#endif
```

```
try(
        sigwait(&sigset, &sig), (!0)
)
        /*     Send SIGALRM signal to connection manager threads     */
try(
        pthread_kill(internet_mngr_tid, SIGALRM), (!0)
)
try(
        pthread_kill(internal_mngr_tid, SIGALRM), (!0)
)
        /*     Wait for threads return    */
try(
        pthread_join(internet_mngr_tid, NULL), (!0)
)
try(
        pthread_join(internal_mngr_tid, NULL), (!0)
)
#ifdef _DEBUG
        syslog(LOG_DEBUG, "Main thread:\tConnection manager threads joined");
#endif
try(
        concurrent_flag_unset(server_status_flag), (1)
)
try(
        pthread_join(joiner_tid, NULL), (!0)
)
#ifdef _DEBUG
        syslog(LOG_DEBUG, "Main thread:\tAll thread joined");
#endif
        /*     Free queue and flag */
try(
        concurrent_queue_destroy(request_queue), (1)
)
try(
        concurrent_flag_destroy(server_status_flag), (1)
)
        /*     Close connections and database    */
try(
        connection_close(internet_connection), (-1)
)
try(
        connection_close(internal_connection), (-1)
)
#ifdef _DEBUG
        syslog(LOG_DEBUG, "Main thread:\tClosed connections");
#endif
try(
        database_close(database), (!0)
)
#ifdef _DEBUG
        syslog(LOG_DEBUG, "Main thread:\tClosed database");
#endif
        syslog(LOG_INFO, "Service stopped");
        return 0;
}

static int daemonize() {
        pid_t pid;    //process id
        pid_t sid;    //session id
        char* wdir;   //working directory
        /* run process in backgound */
        if ((pid = fork()) == -1) {
                return 1;
        }
```

```c
        if (pid != 0) {
                exit(EXIT_SUCCESS);
        }
        /* close standars stream */
        if (fclose(stdin) == EOF) {
                return 1;
        }
        if (fclose(stdout) == EOF) {
                return 1;
        }
        if (fclose(stderr) == EOF) {
                return 1;
        }
        /* create a new session where process is group leader */
        if ((sid = setsid()) == -1) {
                return 1;
        }
        /* fork and kill group leader, lose control of terminal */
        if ((pid = fork()) == -1) {
                return 1;
        }
        if (pid != 0) {
                exit(EXIT_SUCCESS);
        }
        /* change working directory */
        if (asprintf(&wdir, "%s%s", getenv("HOME"), "/.cinema") == -1) {
                return 1;
        }
        if (chdir(wdir) == -1) {
                return 1;
        }
        if (setenv("PWD", wdir, 1)) {
                return 1;
        }
        free(wdir);
        /* reset umask */
        umask(0);
        return 0;
}

void* thread_joiner(void* arg) {
        concurrent_flag_t server_status_flag = arg;
        int status;
        int queue_empty;
try(
        concurrent_flag_status(server_status_flag, &status), (1)
)
try(
        concurrent_queue_is_empty(request_queue, &queue_empty), (1)
)
        while (status || !queue_empty) {
                while(!queue_empty){
                        struct request_info* info;
try(
                        concurrent_queue_dequeue(request_queue, (void**)&info), (1)
)
try(
                        pthread_join(info->tid, NULL), (!0)
)
try(
                        connection_close(info->connection), (-1)
)
#ifdef _DEBUG
syslog(LOG_DEBUG, "Joiner thread:\tJoined request thread %ul", info->tid);
#endif
                        free(info);
try(
                        concurrent_queue_is_empty(request_queue, &queue_empty), (1)
)
                }
                sleep(1);
try(
                concurrent_flag_status(server_status_flag, &status), (1)
)
try(
                concurrent_queue_is_empty(request_queue, &queue_empty), (1)
)
        }
#ifdef _DEBUG
try(
        concurrent_queue_is_empty(request_queue, &queue_empty), (1)
)
        syslog(LOG_DEBUG, "Joiner thread:\tClosing joiner thread, queue empty: %d",
queue_empty);
#endif
        return NULL;
}

void* thread_timer(void* arg) {
        pthread_t parent_tid = (pthread_t)arg;
#ifdef _DEBUG
        syslog(LOG_DEBUG, "Timer thread:\tSpowned");
#endif
        /*      Capture SIGALRM signal      */
        sigset_t sigalrm;
try(
        sigemptyset(&sigalrm), (-1)
)
try(
        sigaddset(&sigalrm, SIGALRM), (-1)
)
try(
        pthread_sigmask(SIG_UNBLOCK, &sigalrm, NULL), (!0)
)
        /*      Send SIGALRM after TIMEOUT elapsed      */
        sleep(TIMEOUT);
try(
        pthread_kill(parent_tid, SIGALRM), (!0)
)
#ifdef _DEBUG
        syslog(LOG_DEBUG, "Timer thread:\tSended SIGALRM to request thread %ul", parent_tid);
#endif
        /*      Detach thread */
try(
        pthread_detach(parent_tid), (!0)
)
        return NULL;
}

void* connection_mngr(void* arg) {
        connection_t connection = arg;

        /*      Setup SIGALRM signal handler      */
        sigset_t sigalrm;
        struct sigaction sigact;
try(
        sigemptyset(&sigalrm), (-1)
)
```

```
try(
	sigaddset(&sigalrm, SIGALRM), (-1)
)
	sigact.sa_handler = thread_exit;
	sigact.sa_mask = sigalrm;
	sigact.sa_flags = 0;
try(
	sigaction(SIGALRM, &sigact, NULL), (-1)
)
	/*	Capture SIGALRM signal	*/
try(
	pthread_sigmask(SIG_UNBLOCK, &sigalrm, NULL), (!0)
)
	/*	Start listen on connection */
try(
	connection_listen(connection), (-1)
)
#ifdef _DEBUG
	syslog(LOG_DEBUG, "CntMng thread:\tlistening on socket");
#endif
	while (1) {
	/*	Wait for incoming connection	*/
		connection_t accepted_connection;
try(
		accepted_connection = connection_accepted(connection), (NULL)
)
	/*	Ignore SIGALRM signal	*/
try(
		pthread_sigmask(SIG_BLOCK, &sigalrm, NULL), (!0)
)
	/*	Create request handler thread	*/
		struct request_info* accepted_request;
try(
		accepted_request = malloc(sizeof(struct request_info)), (NULL)
)
		accepted_request->connection = accepted_connection;
try(
		pthread_create(&accepted_request->tid, NULL, request_handler,
accepted_connection), (!0)
)
	/*	Register thread in the queue	*/
try(
		concurrent_queue_enqueue(request_queue, (void*)accepted_request), (1)
)
	/*	Capture SIGALRM signal	*/
try(
		pthread_sigmask(SIG_UNBLOCK, &sigalrm, NULL), (!0)
)
	}
}

void* request_handler(void* arg) {
	connection_t connection = arg;
	pthread_t timer_tid;
	char* buff;
	char* msg;
#ifdef _DEBUG
	syslog(LOG_DEBUG, "Request thread:\t%ul spowned", pthread_self());
#endif
	/*	Start timeout thread*/
try(
	pthread_create(&timer_tid, NULL, thread_timer, (void*)pthread_self()), (!0)
)
/*	Capture SIGALRM signal	*/
```

```
        sigset_t sigalrm;
try(
        sigemptyset(&sigalrm), (-1)
)
try(
        sigaddset(&sigalrm, SIGALRM), (-1)
)
try(
        pthread_sigmask(SIG_UNBLOCK, &sigalrm, NULL), (!0)
)
        /*     Get the request     */
try(
        connection_recv(connection, &buff), (-1)
)
        /*     Ignore SIGALRM      and stop timeout thread    */
try(
        pthread_sigmask(SIG_BLOCK, &sigalrm, NULL), (!0)
)
try(
        pthread_kill(timer_tid, SIGALRM), (!0)
)
try(
        pthread_join(timer_tid, NULL), (!0)
)
#ifdef _DEBUG
        syslog(LOG_DEBUG, "Request thread:\tStopped timer thread");
#endif
        /*     Elaborate the response */
try(
        database_execute(database, buff, &msg), (1)
)
        free(buff);
        /*     Send the response   */
try(
        connection_send(connection, msg), (-1 - ((errno == ECONNRESET) + (errno == EPIPE)))
)
        free(msg);
#ifdef _DEBUG
        syslog(LOG_DEBUG, "Request thread:\t%ul ready to exit", pthread_self());
#endif
        return NULL;
}
```

database.h

```
#pragma once

typedef void* database_t;

/*     Create database, return database handle on success or return NULL and set properly
errno on error */

database_t database_init(const char *filename);

/*     Close database, return 0 on success or return 1 and set properly errno on error */

int database_close(const database_t handle);

/*     Execute the query received, set the result parameter, return 0 on success or return 1
and set properly errno on error */

int database_execute(const database_t handle, const char *query, char **result);
```

database.c

```
/*     Simple NoSQL Database      */

#include "database.h"

#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/file.h>
#include <errno.h>

#include "resources.h"
#include "storage.h"

struct cinema_info {
       int rows;
       int columns;
};

struct database {
       storage_t storage;
       struct cinema_info cinema_info;
};

/*     Prototype declarations of functions included in this code module    */

static int parse_query(const char* query, char*** parsed);
static int procedure_populate(const database_t handle, char** result);
static int procedure_setup(const database_t handle, char** result);
static int procedure_clean(const database_t handle, char** result);
static int procedure_get_id(const database_t handle, char** result);
static int procedure_get(const database_t handle, char** query, char** result);
static int procedure_set(const database_t handle, char** query, char** result);
static int procedure_map(const database_t handle, char** query, char** result);
static int procedure_book(const database_t handle, char** query, char** result);
static int procedure_unbook(const database_t handle, char** query, char** result);

database_t database_init(const char* filename) {
       struct database* database;
       if ((database = malloc(sizeof(struct database))) == NULL) {
```

```c
			return NULL;
		}
		if ((database->storage = storage_init(filename)) == NULL) {
			free(database);
			return NULL;
		}
		database->cinema_info.columns = 0;
		database->cinema_info.rows = 0;
		return database;
}

int database_close(const database_t handle) {
		struct database* database = (struct database*)handle;
		if (storage_close(database->storage)) {
			return 1;
		}
		free(database);
		return 0;
}

int database_execute(const database_t handle, const char* query, char** result) {
		struct database* database = (struct database*)handle;
		/*	Parse query	*/
		int ret;
		int n_param;
		char** parsed_query;
		if ((n_param = parse_query(query, &parsed_query)) == -1) {
			return 1;
		}
		else if (n_param == 1 && !strcmp(parsed_query[0], "POPULATE")) {
			ret = procedure_populate(database, result);
		}
		else if (n_param == 1 && !strcmp(parsed_query[0], "SETUP")) {
			ret = procedure_setup(database, result);
		}
		else if (n_param == 1 && !strcmp(parsed_query[0], "CLEAN")) {
			ret = procedure_clean(database, result);
		}
		else if (n_param == 1 && !strcmp(parsed_query[0], "ID")) {
			ret = procedure_get_id(database, result);
		}
		else if (n_param == 2 && !strcmp(parsed_query[0], "GET")) {
			ret = procedure_get(database, &(parsed_query[1]), result);
		}
		else if (n_param == 3 && !strcmp(parsed_query[0], "SET")) {
			ret = procedure_set(database, &(parsed_query[1]), result);
		}
		else if (n_param == 2 && !strcmp(parsed_query[0], "MAP")) {
			ret = procedure_map(database, &(parsed_query[1]), result);
		}
		else if (n_param > 2 && !strcmp(parsed_query[0], "BOOK")) {
			ret = procedure_book(database, &(parsed_query[1]), result);
		}
		else if (n_param > 2 && !strcmp(parsed_query[0], "DELETE")) {
			ret = procedure_unbook(database, &(parsed_query[1]), result);
		}
		else {
			*result = strdup(MSG_FAIL);
			ret = 0;
		}
		free(*parsed_query);
		free(parsed_query);
		return ret;
}
```

```c
/*	Tokenize the received query in a string vector saved in parsed parameter
	return the number of token on success or return -1 and set properly errno on error */

static int parse_query(const char* query, char*** parsed) {
	int ntoken;
	char* buffer = NULL;
	char* saveptr = NULL;
	char** token = NULL;

	ntoken = 0;
	if ((buffer = strdup(query)) == NULL) {
		return -1;
	}
	if ((token = malloc(sizeof(char*))) == NULL) {
		return -1;
	}
	token[0] = strtok_r(buffer, " ", &saveptr);
	ntoken++;
	do {
		ntoken++;
		if ((token = realloc(token, sizeof(char*) * (size_t)ntoken)) == NULL) {
			return -1;
		}
		token[ntoken - 1] = strtok_r(NULL, " ", &saveptr);
	} while (token[ntoken - 1] != NULL);
	ntoken--;
	*parsed = token;
	return ntoken;
}

/*	Populate the database executing a predefined set of query */

static int procedure_populate(const database_t handle, char** result) {
	struct database* database = (struct database*)handle;
	char* msg_init[] = {
	"SET IP 127.0.0.1",
	"SET PORT 55555",
	"SET PID 0",
	"SET TIMESTAMP 0",
	"SET ROWS 1",
	"SET COLUMNS 1",
	"SET FILM Titolo",
	"SET SHOWTIME 00:00",
	"SET ID_COUNTER 0",
	"SET 0 0",
	NULL
	};
	for (int i = 0; msg_init[i]; i++) {
		if (database_execute(database, msg_init[i], result)) {
			return 1;
		}
		free(*result);
	}
	*result = strdup(MSG_SUCC);
	return 0;
}

/*	Setup the database and the database info */

static int procedure_setup(const database_t handle, char** result) {
	struct database* database = (struct database*)handle;
	int clean = 0;
	database_execute(database, "GET ROWS", result);
	if (strtoi(*result, &database->cinema_info.rows)) {
		free(*result);
		return 1;
	}
	free(*result);
	database_execute(database, "GET COLUMNS", result);
	if (strtoi(*result, &database->cinema_info.columns)) {
		free(*result);
		return 1;
	}
	free(*result);
	for (int i = 0; i < database->cinema_info.rows * database->cinema_info.columns; i++)
{
		char* query;
		if (asprintf(&query, "GET %d", i) == -1) {
			return 1;
		}
		if (database_execute(database, query, result) == 1) {
			return 1;
		}
		if (!strncmp(*result, MSG_FAIL, strlen(MSG_FAIL))) {
			clean = 1;
			free(query);
			free(*result);
			if (asprintf(&query, "SET %d 0", i) == -1) {
				return 1;
			}
			if (database_execute(database, query, result) == 1) {
				return 1;
			}
		}
		free(query);
		free(*result);
	}
	if (clean) {
		if (procedure_clean(database, result)) {
			return 1;
		}
		free(*result);
	}
	*result = strdup(MSG_SUCC);
	return 0;
}

/*	Discard all the seats prenotation */

static int procedure_clean(const database_t handle, char** result) {
	struct database* database = (struct database*)handle;
	char** parsed_query;
	for (int i = 0; i < database->cinema_info.rows * database->cinema_info.columns; i++)
{
		char* tmp_query;
		char* buffer;
		if (asprintf(&tmp_query, "%d 0", i) == -1) {
			return 1;
		}
		if (parse_query(tmp_query, &parsed_query) == -1) {
			return 1;
		}
		if (procedure_set(database, parsed_query, &buffer)) {
			return 1;
		}
		free(*parsed_query);
```

```c
			free(parsed_query);
			free(tmp_query);
			free(buffer);
		}
		if (parse_query("ID_COUNTER 0", &parsed_query) == -1) {
			return 1;
		}
		if (procedure_set(database, parsed_query, result)) {
			return 1;
		}
		free(*parsed_query);
		free(parsed_query);
		return 0;
}

/*	Get a valid ID for a booking	*/

static int procedure_get_id(const database_t handle, char** result) {
		struct database* database = (struct database*)handle;
		int current_id;
		char* new_id;
		char* buffer;

		if (storage_lock_exclusive(database->storage, "ID_COUNTER")) {
			return 1;
		}
		if (storage_load(database->storage, "ID_COUNTER", &buffer)) {
			return 1;
		}
		if (strtoi(buffer, &current_id)) {
			free(buffer);
			return 1;
		}
		free(buffer);
		if (asprintf(&new_id, "%d", current_id + 1) == -1) {
			return 1;
		}
		if (storage_store(database->storage, "ID_COUNTER", new_id, &buffer)) {
			free(new_id);
			return 1;
		}
		*result = strdup(new_id);
		free(new_id);
		free(buffer);
		if (storage_unlock(database->storage, "ID_COUNTER")) {
			return 1;
		}
		return 0;
}

/*	Standard get procedure	*/

static int procedure_get(const database_t handle, char** query, char** result) {
		struct database* database = (struct database*)handle;
		if (storage_lock_shared(database->storage, query[0])) {
			return 1;
		}
		if (storage_load(database->storage, query[0], result)) {
			return 1;
		}
		if (storage_unlock(database->storage, query[0])) {
			return 1;
		}
		return 0;
```

```c
}

/*	Standard set procedure	*/

static int procedure_set(const database_t handle, char** query, char** result) {
	struct database* database = (struct database*)handle;
	if (storage_lock_exclusive(database->storage, query[0])) {
		return 1;
	}
	if (storage_store(database->storage, query[0], query[1], result)) {
		return 1;
	}
	if (storage_unlock(database->storage, query[0])) {
		return 1;
	}
	return 0;
}

/*	Return the seats status map*/

static int procedure_map(const database_t handle, char** query, char** result) {
	struct database* database = (struct database*)handle;
	int n_seats;
	int id;
	char* map = NULL;

	n_seats = database->cinema_info.rows * database->cinema_info.columns;
	if (strtoi(query[0], &id)) {
		*result = strdup(MSG_FAIL);
		return 0;
	}
	if (!n_seats) {
		*result = strdup(MSG_FAIL);
		return 0;
	}
	if ((map = malloc(sizeof(char) * (size_t)(n_seats * 2))) == NULL) {
		return 1;
	}
	memset(map, 0, (size_t)(n_seats * 2));
	for (int i = 0; i < n_seats; i++) {
		int book_id;
		char* tmp_query;
		char* buffer;
		if (asprintf(&tmp_query, "%d", i) == -1) {
			return 1;
		}
		if (procedure_get(database, &tmp_query, &buffer)) {
			return 1;
		}
		free(tmp_query);
		if (strtoi(buffer, &book_id)) {
			free(buffer);
			*result = strdup(MSG_FAIL);
			return 0;
		}
		if (book_id) {
			free(buffer);
			if (book_id == id) {
				asprintf(&buffer, "1");
			}
			else {
				asprintf(&buffer, "2");
			}
		}
		map[2 * i] = buffer[0];
		map[(2 * i) + 1] = ' ';
		free(buffer);
	}
	map[(n_seats * 2) - 1] = 0;
	*result = map;
	return 0;
}

/*	Return the ID on a successfull operation */

static int procedure_book(const database_t handle, char** query, char** result) {
	struct database* database = (struct database*)handle;
	int n_seats;
	char* id = query[0];
	char** ordered_request;
	int* tmp;	//tmp variable to order request
	int abort = 0;

	/*	order request to avoid deadlock	*/
	for (n_seats = 0; query[n_seats + 1]; n_seats++);	//get n_seats
	if ((tmp = malloc(sizeof(int) * (size_t)(database->cinema_info.rows * database-
>cinema_info.columns))) == NULL) {
		return 1;
	}
	memset(tmp, 0, sizeof(int) * (size_t)(database->cinema_info.rows * database-
>cinema_info.columns));
	for (int i = 0; i < n_seats; i++) {
		int seat;
		if (strtoi(query[i + 1], &seat)) {
			free(tmp);
			*result = strdup(MSG_FAIL);
			return 0;
		}
		tmp[seat] = 1;
	}
	if ((ordered_request = malloc(sizeof(char*) * (size_t)n_seats)) == NULL) {
		return 1;
	}
	int n_tmp = 0;
	for (int i = 0; i < database->cinema_info.rows * database->cinema_info.columns; i++)
{
		if (tmp[i]) {
			asprintf(&(ordered_request[n_tmp++]), "%d", i);
			if (n_tmp == n_seats) {
				break;
			}
		}
	}
	free(tmp);
	if (n_tmp != n_seats) {
		*result = strdup(MSG_FAIL);
		return 0;
	}
	/*	2PL locking	*/
	for (int i = 0; i < n_seats; i++) {
		if (storage_lock_exclusive(database->storage, ordered_request[i])) {
			return 1;
		}
	}
	char* seat_id;
	for (int i = 0; i < n_seats; i++) {
		if (storage_load(database->storage, ordered_request[i], &seat_id)) {
			return 1;
```

```c
                }
                if (strcmp(seat_id, "0")) {
                        abort = 1;
                        free(seat_id);
                        break;
                }
                free(seat_id);
        }
        if (!abort) {
                if (!strcmp(id, "-1")) {
                        if (database_execute(database, "ID", &id)) {
                                return 1;
                        }
                }
                for (int i = 0; i < n_seats; i++) {
                        char* buffer;
                        if (storage_store(database->storage, ordered_request[i], id, &buffer))
{
                                return 1;
                        }
                        free(buffer);
                }
                *result = strdup(id);
        }
        else {
                *result = strdup(MSG_FAIL);
        }
        for (int i = 0; i < n_seats; i++) {
                if (storage_unlock(database->storage, ordered_request[i])) {
                        return 1;
                }
        }
        for (int i = 0; i < n_seats; i++) {
                free(ordered_request[i]);
        }
        free(ordered_request);
        return 0;
}

/*      Remove a booking      */

static int procedure_unbook(const database_t handle, char** query, char** result) {
        struct database* database = (struct database*)handle;
        char* id = query[0];

        for (int i = 1; query[i]; i++) {
                char* buffer;
                if (procedure_get(database, &(query[i]), &buffer)) {
                        return 1;
                }
                if (strcmp(id, buffer)) {
                        free(buffer);
                        *result = strdup(MSG_FAIL);
                        return 0;
                }
                free(buffer);
        }
        for (int i = 1; query[i]; i++) {
                char** parsed_query;
                char* tmp_query;
                char* buffer;
                if (asprintf(&tmp_query, "%s 0", query[i]) == -1) {
                        return 1;
                }
```

```
            if (parse_query(tmp_query, &parsed_query) == -1) {
                return 1;
            }
            if (procedure_set(database, parsed_query, &buffer)) {
                return 1;
            }
            free(*parsed_query);
            free(parsed_query);
            free(tmp_query);
            free(buffer);
        }
        *result = strdup(MSG_SUCC);
        return 0;
}
```

storage.h

```c
#pragma once

#define MSG_SUCC "OPERATION SUCCEDED"
#define MSG_FAIL "OPERATION FAILED"

typedef void* storage_t;

/*     Create storage, return database handle on success or return NULL and set properly
errno on error */

storage_t storage_init(const char* filename);

/*     Close the storage, return 0 on succes or return 1 and set properly errno on error */

int storage_close(const storage_t handle);

/*     Store the value linked to the key in the storage */

int storage_store(const storage_t handle, const char* key, const char* value, char**
result);

/*     Load the value linked to the key from the storage */

int storage_load(const storage_t handle, const char* key, char** result);

/*     Llock as shared the lock linked to the key */

int storage_lock_shared(const storage_t handle, const char* key);

/*     Llock as exclusive the lock linked to the key */

int storage_lock_exclusive(const storage_t handle, const char* key);

/*     Unlock the lock linked to the key */

int storage_unlock(const storage_t handle, const char* key);
```

storage.c

```
#include "storage.h"

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <errno.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/file.h>
#include <string.h>

#include "index_table.h"

#define MAXLEN 16

struct index_record {
	long offset;
	pthread_rwlock_t lock;
};

struct storage {
	FILE* stream;
	char* buffer_cache;
	index_table_t index_table;
	pthread_mutex_t mutex_seek_stream;
	pthread_rwlock_t lock_buffer_cahce;
};

/*	Prototype declarations of functions included in this code module	*/

static int lexicographical_comparison(const void* key1, const void* key2);
static int update_buffer_cache(const storage_t handle);
static int load_table(const storage_t handle);
static int format(const char* str, char** result);
static index_record_t record_init();
static int record_destroy(void* key, void* value);

/*	Initialize the storage,	*/

storage_t storage_init(const char* filename) {
	struct storage* storage;
	if ((storage = malloc(sizeof(struct storage))) == NULL) {
		return NULL;
	}
	if ((storage->stream = fopen(filename, "r+")) == NULL) {
		free(storage);
		return NULL;
	}
	if (flock(fileno(storage->stream), LOCK_EX | LOCK_NB) == -1) {	//instead of 
semget & ftok to avoid mix SysV and POSIX, replace with fcntl
		free(storage);
		return NULL;
	}
	storage->buffer_cache = NULL;
	int ret;
	while ((ret = pthread_mutex_init(&storage->mutex_seek_stream, NULL)) && errno == 
EINTR);
	if (ret) {
		free(storage);
		return NULL;
	}
	while ((ret = pthread_rwlock_init(&storage->lock_buffer_cahce, NULL)) && errno ==
EINTR);
	if (ret) {
		free(storage);
		return NULL;
	}
	if ((storage->index_table = index_table_init(&record_init, &record_destroy,
&lexicographical_comparison)) == NULL) {
		free(storage);
		return NULL;
	}
	if (load_table(storage)) {
		free(storage);
		return NULL;
	}
	if (update_buffer_cache(storage)) {
		free(storage);
		return NULL;
	}
	return storage;
}

int storage_close(const storage_t handle) {
	struct storage* storage = (struct storage*)handle;
	int ret;
	while ((ret = pthread_mutex_destroy(&storage->mutex_seek_stream)) && errno == EINTR);
	if (ret) {
		return 1;
	}
	while ((ret = pthread_rwlock_destroy(&storage->lock_buffer_cahce)) && errno ==
EINTR);
	if (ret) {
		return 1;
	}
	index_table_destroy(storage->index_table);
	fclose(storage->stream);
	free(storage->buffer_cache);
	free(storage);
	return 0;
}

int storage_store(const storage_t handle, const char* key, const char* value, char** result)
{
	struct storage* storage = (struct storage*)handle;
	struct index_record* record;
	int ret;
	char* formatted_key;
	char* formatted_value;
	if (format(key, &formatted_key)) {
		return 1;
	}
	if (format(value, &formatted_value)) {
		return 1;
	}
	if (formatted_key == NULL) {
		*result = strdup(MSG_FAIL);
		return 0;
	}
	if (formatted_value == NULL) {
		*result = strdup(MSG_FAIL);
		return 0;
	}
	if ((record = index_table_search(storage->index_table, strdup(formatted_key))) ==
NULL) {
		return 1;
	}
	/*	add record if it doesn't exist	*/
	if (record->offset == -1) {
		while ((ret = pthread_rwlock_wrlock(&storage->lock_buffer_cahce)) && errno ==
EINTR);
		if (ret) {
			return 1;
		}
		while ((ret = pthread_mutex_lock(&storage->mutex_seek_stream)) && errno ==
EINTR);
		if (ret) {
			return 1;
		}
		long offset;
		if (fseek(storage->stream, 0, SEEK_END) == -1) {
			return 1;
		}
		if ((offset = ftell(storage->stream)) == -1) {
			return 1;
		}
		for (int i = 0; i < MAXLEN; i++) {
			if (fputc(formatted_key[i], storage->stream) == EOF) {
				return 1;
			}
		}
		for (int i = 0; i < MAXLEN; i++) {
			if (fputc(0, storage->stream) == EOF) {
				return 1;
			}
		}
		fflush(storage->stream);
		while ((ret = pthread_mutex_unlock(&storage->mutex_seek_stream)) && errno ==
EINTR);
		if (ret) {
			return 1;
		}
		record->offset = offset + MAXLEN;
		update_buffer_cache(storage);
		while ((ret = pthread_rwlock_unlock(&storage->lock_buffer_cahce)) && errno ==
EINTR);
		if (ret) {
			return 1;
		}
	}
	free(formatted_key);
	while ((ret = pthread_mutex_lock(&storage->mutex_seek_stream)) && errno == EINTR);
	if (ret) {
		return 1;
	}
	if (fseek(storage->stream, record->offset, SEEK_SET) == -1) {
		return 1;
	}
	for (int i = 0; i < MAXLEN; i++) {
		if (fputc(formatted_value[i], storage->stream) == EOF) {
			return 1;
		}
	}
	fflush(storage->stream);
	while ((ret = pthread_mutex_unlock(&storage->mutex_seek_stream)) && errno == EINTR);
	if (ret) {
		return 1;
	}
	for (int i = 0; i < MAXLEN; i++) {
```

```c
			storage->buffer_cache[record->offset + i] = formatted_value[i];
		}
		free(formatted_value);
		*result = strdup(MSG_SUCC);
		return 0;
}

int storage_load(const storage_t handle, const char* key, char** result) {
		struct storage* storage = (struct storage*)handle;
		struct index_record* record;
		int ret;
		char* formatted_key;
		if (format(key, &formatted_key)) {
			return 1;
		}
		if (formatted_key == NULL) {
			*result = strdup(MSG_FAIL);
			return 0;
		}
		if ((record = index_table_search(storage->index_table, strdup(formatted_key))) ==
NULL) {
			return 1;
		}
		free(formatted_key);
		if (record->offset == -1) {
			index_table_delete(storage->index_table, key);
			*result = strdup(MSG_FAIL);
			return 0;
		}
		if ((*result = malloc(sizeof(char) * MAXLEN + 1)) == NULL) {
			return 1;
		}
		while ((ret = pthread_rwlock_rdlock(&storage->lock_buffer_cahce)) && errno == EINTR);
		if (ret) {
			return 1;
		}
		for (int i = 0; i < MAXLEN; i++) {
			(*result)[i] = storage->buffer_cache[record->offset + i];
		}
		(*result)[MAXLEN] = 0;
		while ((ret = pthread_rwlock_unlock(&storage->lock_buffer_cahce)) && errno == EINTR);
		if (ret) {
			return 1;
		}
		return 0;
}

int storage_lock_shared(const storage_t handle, const char* key) {
		struct storage* storage = (struct storage*)handle;
		struct index_record* record;
		int ret;
		char* formatted_key;
		if (format(key, &formatted_key)) {
			return 1;
		}
		if (formatted_key == NULL) {
			return 0;
		}
		if ((record = index_table_search(storage->index_table, strdup(formatted_key))) ==
NULL) {
			return 1;
		}
		free(formatted_key);
		while ((ret = pthread_rwlock_rdlock(&record->lock)) && errno == EINTR);
```

```c
	if (ret) {
		return 1;
	}
	return 0;
}

int storage_lock_exclusive(const storage_t handle, const char* key) {
	struct storage* storage = (struct storage*)handle;
	struct index_record* record;
	int ret;
	char* formatted_key;
	if (format(key, &formatted_key)) {
		return 1;
	}
	if (formatted_key == NULL) {
		return 0;
	}
	if ((record = index_table_search(storage->index_table, strdup(formatted_key))) ==
NULL) {
		return 1;
	}
	free(formatted_key);
	while ((ret = pthread_rwlock_wrlock(&record->lock)) && errno == EINTR);
	if (ret) {
		return 1;
	}
	return 0;
}

int storage_unlock(const storage_t handle, const char* key) {
	struct storage* storage = (struct storage*)handle;
	struct index_record* record;
	int ret;
	char* formatted_key;
	if (format(key, &formatted_key)) {
		return 1;
	}
	if (formatted_key == NULL) {
		return 0;
	}
	if ((record = index_table_search(storage->index_table, strdup(formatted_key))) ==
NULL) {
		return 1;
	}
	free(formatted_key);
	if (record->offset == -1) {
		index_table_delete(storage->index_table, key);
		return 0;
	}
	while ((ret = pthread_rwlock_unlock(&record->lock)) && errno == EINTR);
	if (ret) {
		return 1;
	}
	return 0;
}

/*	Function to compare two strings	*/


static int lexicographical_comparison(const void* key1, const void* key2) {
	char* str1 = (char*)key1;
	char* str2 = (char*)key2;
	for (int i = 0; ; i++) {
		if (!str1[i] && !str2[i]) {
				return 0;
			}
			else if (str1[i] && !str2[i]) {
				return 1;
			}
			else if (!str1[i] && str2[i]) {
				return -1;
			}
			else if (str1[i] < str2[i]) {
				return -1;
			}
			else if (str1[i] > str2[i]) {
				return 1;
			}
		}
}

/*	Initialize the value in the index table depending on the content od the storage */

static int load_table(const storage_t handle) {
	struct storage* storage = (struct storage*)handle;
	if (fseek(storage->stream, 0, SEEK_SET) == -1) {
		return 1;
	}
	while ((fgetc(storage->stream)) != EOF) {
		if (fseek(storage->stream, -1, SEEK_CUR) == -1) {
			return 1;
		}
		char* current_key;
		struct index_record* current_record;
		if ((current_key = malloc(sizeof(char) * (MAXLEN + 1))) == NULL) {
			return 1;
		}
		for (int i = 0; i < MAXLEN; i++) {
			if ((int)(current_key[i] = (char)fgetc(storage->stream)) == EOF) {
				free(current_key);
				return 1;
			}
		}
		current_key[MAXLEN] = 0;
		if ((current_record = record_init()) == NULL) {
			free(current_key);
			return 1;
		}
		if ((current_record->offset = ftell(storage->stream)) == -1) {
			free(current_key);
			free(current_record);
			return 1;
		}
		if (index_table_insert(storage->index_table, current_key, current_record)) {
			free(current_key);
			free(current_record);
			return 1;
		}
		if (fseek(storage->stream, MAXLEN, SEEK_CUR) == -1) {
			free(current_key);
			free(current_record);
			return 1;
		}
	}
	if (!feof(storage->stream)) {
		return 1;
	}
	return 0;
```

```c
}

/*	Update the buffer cache */

static int update_buffer_cache(const storage_t handle) {
	struct storage* storage = (struct storage*)handle;
	long filesize;
	int ret;

	free(storage->buffer_cache);
	while ((ret = pthread_mutex_lock(&storage->mutex_seek_stream)) && errno == EINTR);
	if (ret) {
		return 1;
	}
	if (fseek(storage->stream, 0, SEEK_END) == -1) {
		return 1;
	}
	if ((filesize = ftell(storage->stream)) == -1) {
		return 1;
	}
	if ((storage->buffer_cache = malloc(sizeof(char) * (size_t)(filesize + 1))) == NULL)
{
		return 1;
	}
	if (fseek(storage->stream, 0, SEEK_SET) == -1) {
		return 1;
	}
	if (fgets(storage->buffer_cache, (int)filesize + 1, storage->stream) == NULL) {
		return 1;
	}
	while ((ret = pthread_mutex_unlock(&storage->mutex_seek_stream)) && errno == EINTR);
	if (ret) {
		return 1;
	}
	return 0;
}

/*	Format the received string in a valid string saved in the result parameter,
	return 0 on success or return 1 and set properly errno on error */

static int format(const char* str, char** result) {
	if (strlen(str) > MAXLEN) {
		*result = NULL;
		return 0;
	}
	if ((*result = malloc(sizeof(char) * MAXLEN + 1)) == NULL) {
		return 1;
	}
	memset(*result, 0, MAXLEN + 1);
	strncpy(*result, str, MAXLEN);
	return 0;
}

/*	Function to create a record data type return NULL on failure and set properly errno
on error	*/

static index_record_t record_init() {
	struct index_record* record;
	if ((record = malloc(sizeof(struct index_record))) == NULL) {
		return NULL;
	}
	record->offset = -1;
	int ret;
	while ((ret = pthread_rwlock_init(&record->lock, NULL)) && errno == EINTR);
	if (ret) {
		free(record);
		return NULL;
	}
	return record;
}

/*	Function to destroy a record data type return 1 on failure and set properly errno on
error	*/

static int record_destroy(void* key, void* value) {
	int ret;
	struct index_record* record = (struct index_record*)value;
	while ((ret = pthread_rwlock_destroy(&record->lock)) && errno == EINTR);
	if (ret) {
		return 1;
	}
	free(key);
	free(record);
	return 0;
}
```

Index_table.h

```
#pragma once

typedef void* index_record_t;
typedef void* index_table_t;

index_table_t index_table_init(
    const index_record_t (*record_init)(),
    const int (*record_destroy)(void* key, void* value),
    const int (*comparison_function)(const void* key1, const void* key2));

int index_table_destroy(const index_table_t handle);

int index_table_insert(const index_table_t handle, const void* key, const void* record);

int index_table_delete(const index_table_t handle, const void* key);

index_record_t index_table_search(const index_table_t handle, void* key);
```

index_table.c

```c
#include "index_table.h"

#include <stdlib.h>
#include <pthread.h>
#include <errno.h>

#include "avl_tree.h"
#include "stack.h"

struct index_table {
	avl_tree_t avl_tree;
	pthread_rwlock_t lock;
	index_record_t (*record_init)();
	int (*record_destroy)(void* key, void* value);
};

index_table_t index_table_init(
	const index_record_t(*record_init)(),
	const int (*record_destroy)(void* key, void* value),
	const int (*comparison_function)(const void* key1, const void* key2)) {

	struct index_table* index_table;
	if ((index_table = malloc(sizeof(struct index_table))) == NULL) {
		return NULL;
	}
	if ((index_table->avl_tree = avl_tree_init(comparison_function)) == NULL) {
		free(index_table);
		return NULL;
	}
	int ret;
	while ((ret = pthread_rwlock_init(&index_table->lock, NULL)) && errno == EINTR);
	if (ret) {
		free(index_table);
		return NULL;
	}
	index_table->record_init = record_init;
	index_table->record_destroy = record_destroy;
	return index_table;
}

int index_table_destroy(const index_table_t handle) {
	struct index_table* index_table = (struct index_table*)handle;
	int ret;
	while ((ret = pthread_rwlock_destroy(&index_table->lock)) && errno == EINTR);
	if (ret) {
		return 1;
	}
	avl_tree_node_t node = avl_tree_get_root(index_table->avl_tree);
	_stack_t stack = stack_init();
	if (node) {
		stack_push(stack, node);
		while (!stack_is_empty(stack)) {
			struct binary_tree_node* current_node;
			current_node = stack_pop(stack);
			if (avl_tree_node_get_left_son(current_node)) {
				stack_push(stack, avl_tree_node_get_left_son(current_node));
			}
			if (avl_tree_node_get_right_son(current_node)) {
				stack_push(stack, avl_tree_node_get_right_son(current_node));
			}
```

```
				if (index_table->record_destroy(avl_tree_node_get_key(current_node),
avl_tree_node_get_value(current_node))) {
					return 1;
				}
			}
		}
		stack_destroy(stack);
		if (avl_tree_destroy(index_table->avl_tree)) {
			return 1;
		}
		free(index_table);
		return 0;
}

int index_table_insert(const index_table_t handle, const void* key, const void* record) {
		struct index_table* index_table = (struct index_table*)handle;
		int result;
		int ret;
		while ((ret = pthread_rwlock_wrlock(&index_table->lock)) && errno == EINTR);
		if (ret) {
			return 1;
		}
		result = avl_tree_insert(index_table->avl_tree, key, record);
		while ((ret = pthread_rwlock_unlock(&index_table->lock)) && errno == EINTR);
		if (ret) {
			return 1;
		}
		return result;
}

int index_table_delete(const index_table_t handle, const void* key) {
		struct index_table* index_table = (struct index_table*)handle;
		int result;
		int ret;
		while ((ret = pthread_rwlock_wrlock(&index_table->lock)) && errno == EINTR);
		if (ret) {
			return 1;
		}
		avl_tree_node_t node = avl_tree_search_node(index_table->avl_tree, key);
		if (index_table->record_destroy(avl_tree_node_get_key(node),
avl_tree_node_get_value(node))) {
			return 1;
		}
		result = avl_tree_delete_node(index_table->avl_tree, node);
		while ((ret = pthread_rwlock_unlock(&index_table->lock)) && errno == EINTR);
		if (ret) {
			return 1;
		}
		return result;
}

index_record_t index_table_search(const index_table_t handle, void* key) {
		struct index_table* index_table = (struct index_table*)handle;
		void* result;
		int ret;
		while ((ret = pthread_rwlock_rdlock(&index_table->lock)) && errno == EINTR);
		if (ret) {
			return NULL;
		}
		if ((result = avl_tree_search(index_table->avl_tree, key)) == NULL) {
			while ((ret = pthread_rwlock_unlock(&index_table->lock)) && errno == EINTR);
			if (ret) {
				return NULL;
			}
```

```
            while ((ret = pthread_rwlock_wrlock(&index_table->lock)) && errno == EINTR);
            if (ret) {
                return NULL;
            }
            if ((result = avl_tree_search(index_table->avl_tree, key)) == NULL) {
                index_record_t record;
                if ((record = index_table->record_init()) == NULL) {
                    return NULL;
                }
                if (avl_tree_insert(index_table->avl_tree, key, record)) {
                    return NULL;
                }
                result = record;
            }
        }
        else {
            free(key);
        }
        while ((ret = pthread_rwlock_unlock(&index_table->lock)) && errno == EINTR);
        if (ret) {
            return NULL;
        }
        return result;
    }
```

avl_tree.h

```c
#pragma once

#include "avl_tree_node.h"

typedef void* avl_tree_t;

typedef int avl_tree_comparison_function(const void* key1, const void* key2);

avl_tree_t avl_tree_init(const avl_tree_comparison_function* comparison_function);

int avl_tree_destroy(const avl_tree_t handle);

long avl_tree_nodes_number(const avl_tree_t handle);

avl_tree_node_t avl_tree_get_root(const avl_tree_t handle);

void avl_tree_set_root(const avl_tree_t handle, avl_tree_node_t node);

void avl_tree_insert_as_left_subtree(const avl_tree_t handle, avl_tree_node_t node, const
avl_tree_t subtree);

void avl_tree_insert_as_right_subtree(const avl_tree_t handle, avl_tree_node_t node, const
avl_tree_t subtree);

avl_tree_t avl_tree_cut(const avl_tree_t handle, avl_tree_node_t node);

avl_tree_t avl_tree_cut_left(const avl_tree_t handle, avl_tree_node_t node);

avl_tree_t avl_tree_cut_right(const avl_tree_t handle, avl_tree_node_t node);

avl_tree_t avl_tree_cut_one_son_node(const avl_tree_t handle, avl_tree_node_t node);

void avl_tree_insert_single_node_tree(const avl_tree_t handle, const void* key, const
avl_tree_t new_tree);

avl_tree_node_t avl_tree_search_node(const avl_tree_t handle, const void* key);

void* avl_tree_search(const avl_tree_t handle, const void* key);

int avl_tree_insert(const avl_tree_t handle, const void* key, const void* value);

int avl_tree_delete(const avl_tree_t handle, const void* key);

int avl_tree_delete_node(const avl_tree_t handle, avl_tree_node_t node);
```

avl_tree.c

```c
#include "avl_tree.h"

#include <stdlib.h>

#include "stack.h"

struct avl_tree {
	avl_tree_node_t root;
	int n;
	avl_tree_comparison_function* compare;
};

static void right_rotation(const avl_tree_t handle, const avl_tree_node_t node);
static void left_rotation(const avl_tree_t handle, const avl_tree_node_t node);
static void rotate(const avl_tree_t handle, const avl_tree_node_t node);
static void balance_insert(const avl_tree_t handle, const avl_tree_node_t node);
static void balance_delete(const avl_tree_t handle, const avl_tree_node_t node);
static int cut_single_son(const avl_tree_t handle, const avl_tree_node_t node);
static int subtree_nodes_number(const avl_tree_t node);
static int defualt_comparison_function(const void* key1, const void* key2);

avl_tree_t avl_tree_init(const avl_tree_comparison_function* comparison_function) {
	struct avl_tree* tree;
	if ((tree = malloc(sizeof(struct avl_tree))) == NULL) {
		return NULL;
	}
	tree->root = NULL;
	tree->n = 0;
	tree->compare = comparison_function ? comparison_function :
&defualt_comparison_function;
	return tree;
}

int avl_tree_destroy(const avl_tree_t handle) {
	struct avl_tree* tree = (struct avl_tree*)handle;
	struct avl_tree* left_subtree;
	struct avl_tree* right_subtree;
	left_subtree = avl_tree_cut_left(tree, tree->root);
	if (!left_subtree) {
		return 1;
	}
	right_subtree = avl_tree_cut_right(tree, tree->root);
	if (!right_subtree) {
		return 1;
	}
	if (left_subtree->root) {
		if (avl_tree_destroy(left_subtree)) {
			return 1;
		}
	}
	else {
		free(left_subtree);
	}
	if (right_subtree->root) {
		if (avl_tree_destroy(right_subtree)) {
			return 1;
		}
	}
	else {
		free(right_subtree);
	}
	if (avl_tree_node_destroy(tree->root)) {
			return 1;
		}
		free(tree);
		return 0;
}

inline long avl_tree_nodes_number(const avl_tree_t handle) {
		struct avl_tree* tree = (struct avl_tree*)handle;
		return tree->n;
}

inline avl_tree_node_t avl_tree_get_root(const avl_tree_t handle) {
		struct avl_tree* tree = (struct avl_tree*)handle;
		return tree->root;
}

inline void avl_tree_set_root(const avl_tree_t handle, const avl_tree_node_t node) {
		struct avl_tree* tree = (struct avl_tree*)handle;
		tree->root = node;
		tree->n = subtree_nodes_number(node);
}

void avl_tree_insert_as_left_subtree(const avl_tree_t handle, const avl_tree_node_t node,
const avl_tree_t subtree) {
		struct avl_tree* tree = (struct avl_tree*)handle;
		struct avl_tree* avl_tree_subtree = (struct avl_tree*)subtree;
		if (avl_tree_subtree->root) {
			avl_tree_node_set_father(avl_tree_subtree->root, node);
		}
		avl_tree_node_set_left_son(node, avl_tree_subtree->root);
		tree->n += subtree_nodes_number(avl_tree_subtree->root);
}

void avl_tree_insert_as_right_subtree(const avl_tree_t handle, const avl_tree_node_t node,
const avl_tree_t subtree) {
		struct avl_tree* tree = (struct avl_tree*)handle;
		struct avl_tree* avl_tree_subtree = (struct avl_tree*)subtree;
		if (avl_tree_subtree->root) {
			avl_tree_node_set_father(avl_tree_subtree->root, node);
		}
		avl_tree_node_set_right_son(node, avl_tree_subtree->root);
		tree->n += subtree_nodes_number(avl_tree_subtree->root);
}

avl_tree_t avl_tree_cut(const avl_tree_t handle, const avl_tree_node_t node) {
		struct avl_tree* tree = (struct avl_tree*)handle;
		if (node) {
			avl_tree_node_t father = avl_tree_node_get_father(node);

			if (node == avl_tree_get_root(tree)) {
				tree->root = NULL;
				tree->n = 0;
			}
			else if (node == avl_tree_node_get_left_son(father)) {
				if (avl_tree_node_degree(node) == 0) {
					avl_tree_node_set_left_son(father, NULL);
					tree->n--;
				}
				else {
					return avl_tree_cut_left(tree, node);
				}
			}
			else if (node == avl_tree_node_get_right_son(father)) {
				if (avl_tree_node_degree(node) == 0) {
```

```
                        avl_tree_node_set_right_son(father, NULL);
                        tree->n--;
                    }
                    else {
                        return avl_tree_cut_right(tree, node);
                    }
                }
        }
        avl_tree_t cutted = avl_tree_init(tree->compare);
        if (cutted) {
                avl_tree_set_root(cutted, node);
        }
        return cutted;
}

avl_tree_t avl_tree_cut_left(const avl_tree_t handle, const avl_tree_node_t father) {
        struct avl_tree* tree = (struct avl_tree*)handle;
        avl_tree_t new_tree;
        avl_tree_node_t son = avl_tree_node_get_left_son(father);
        if ((new_tree = avl_tree_init(tree->compare)) == NULL) {
                return NULL;
        }
        avl_tree_set_root(new_tree, son);
        tree->n -= subtree_nodes_number(son);
        avl_tree_node_set_left_son(father, NULL);
        return new_tree;
}

avl_tree_t avl_tree_cut_right(const avl_tree_t handle, const avl_tree_node_t father) {
        struct avl_tree* tree = (struct avl_tree*)handle;
        avl_tree_t new_tree;
        avl_tree_node_t son = avl_tree_node_get_right_son(father);
        if ((new_tree = avl_tree_init(tree->compare)) == NULL) {
                return NULL;
        }
        avl_tree_set_root(new_tree, son);
        tree->n -= subtree_nodes_number(son);
        avl_tree_node_set_right_son(father, NULL);
        return new_tree;
}

avl_tree_t avl_tree_cut_one_son_node(const avl_tree_t handle, const avl_tree_node_t node) {
        struct avl_tree* tree = (struct avl_tree*)handle;
        avl_tree_t cutted_tree;
        avl_tree_node_t son = NULL;
        if (avl_tree_node_get_left_son(node)) {
                son = avl_tree_node_get_left_son(node);
        }
        else if (avl_tree_node_get_right_son(node)) {
                son = avl_tree_node_get_right_son(node);
        }
        if (!son) {             //node is a leaf
                cutted_tree = avl_tree_cut(tree, node);
        }
        else {
                avl_tree_node_swap(node, son);
                cutted_tree = avl_tree_cut(tree, node);
                avl_tree_insert_as_left_subtree(tree, node, avl_tree_cut_left(cutted_tree,
son));
                avl_tree_insert_as_right_subtree(tree, node, avl_tree_cut_right(cutted_tree,
son));
                //memory leack
        }
        return cutted_tree;
```

```c
}

void avl_tree_insert_single_node_tree(const avl_tree_t handle, const void* key, const
avl_tree_t new_tree) {
	struct avl_tree* tree = (struct avl_tree*)handle;
	int cmp;
	if (avl_tree_get_root(tree) == NULL) {
		avl_tree_set_root(tree, avl_tree_get_root(new_tree));
		//memory leack
	}
	else {
		avl_tree_node_t current = avl_tree_get_root(tree);
		avl_tree_node_t prev = NULL;
		while (current) {
			prev = current;
			cmp = tree->compare(key, avl_tree_node_get_key(current));
			if (cmp < 1) {
				current = avl_tree_node_get_left_son(current);
			}
			else {
				current = avl_tree_node_get_right_son(current);
			}
		}
		cmp = tree->compare(key, avl_tree_node_get_key(prev));
		if (cmp < 1) {
			avl_tree_insert_as_left_subtree(tree, prev, new_tree);
		}
		else {
			avl_tree_insert_as_right_subtree(tree, prev, new_tree);
		}
	}
}

avl_tree_node_t avl_tree_search_node(const avl_tree_t handle, const void* key) {
	struct avl_tree* tree = (struct avl_tree*)handle;
	avl_tree_node_t current = avl_tree_get_root(tree);
	int cmp;
	while (current) {
		cmp = tree->compare(key, avl_tree_node_get_key(current));
		if (!cmp) {
			return current;
		}
		else if (cmp == -1) {
			current = avl_tree_node_get_left_son(current);
		}
		else {
			current = avl_tree_node_get_right_son(current);
		}
	}
	return NULL;
}

inline void* avl_tree_search(const avl_tree_t handle, const void* key) {
	struct avl_tree* tree = (struct avl_tree*)handle;
	return avl_tree_node_get_value(avl_tree_search_node(tree, key));
}

int avl_tree_insert(const avl_tree_t handle, const void* key, const void* value) {
	struct avl_tree* tree = (struct avl_tree*)handle;
	avl_tree_node_t node = avl_tree_node_init(key, value);
	avl_tree_t new_tree = avl_tree_init(tree->compare);
	avl_tree_set_root(new_tree, node);
	avl_tree_insert_single_node_tree(tree, key, new_tree);
	balance_insert(tree, node);
	avl_tree_set_root(new_tree, NULL);
	avl_tree_destroy(new_tree);
	return 0;
}

int avl_tree_delete(const avl_tree_t handle, const void* key) {
	struct avl_tree* tree = (struct avl_tree*)handle;
	avl_tree_node_t node = avl_tree_search_node(tree, key);
	if (node) {
		if (avl_tree_node_degree(node) < 2) {
			return cut_single_son(tree, node);
		}
		else {
			avl_tree_node_t pred = avl_tree_node_get_pred(node);
			avl_tree_node_swap(node, pred);
			int tmp_h = avl_tree_node_get_height(node);
			avl_tree_node_set_height(node, avl_tree_node_get_height(pred));
			avl_tree_node_set_height(pred, tmp_h);
			return cut_single_son(tree, pred);
		}
	}
	return 0;
}

int avl_tree_delete_node(const avl_tree_t handle, const avl_tree_node_t node) {
	struct avl_tree* tree = (struct avl_tree*)handle;
	if (node) {
		if (avl_tree_node_degree(node) < 2) {
			return cut_single_son(tree, node);
		}
		else {
			avl_tree_node_t pred = avl_tree_node_get_pred(node);
			avl_tree_node_swap(node, pred);
			int tmp_h = avl_tree_node_get_height(node);
			avl_tree_node_set_height(node, avl_tree_node_get_height(pred));
			avl_tree_node_set_height(pred, tmp_h);
			return cut_single_son(tree, pred);
		}
	}
	return 0;
}

static void right_rotation(const avl_tree_t handle, const avl_tree_node_t node) {
	struct avl_tree* tree = (struct avl_tree*)handle;
	avl_tree_node_t left_son = avl_tree_node_get_left_son(node);
	avl_tree_node_swap(node, left_son);

	avl_tree_t l_tree = avl_tree_cut_left(tree, node);
	avl_tree_t r_tree = avl_tree_cut_right(tree, node);
	avl_tree_t l_tree_l = avl_tree_cut_left(l_tree, left_son);
	avl_tree_t l_tree_r = avl_tree_cut_right(l_tree, left_son);

	avl_tree_insert_as_right_subtree(l_tree, avl_tree_get_root(l_tree), r_tree);
	avl_tree_insert_as_left_subtree(l_tree, avl_tree_get_root(l_tree), l_tree_r);
	avl_tree_insert_as_right_subtree(tree, node, l_tree);
	avl_tree_insert_as_left_subtree(tree, node, l_tree_l);

	avl_tree_node_update_height(avl_tree_node_get_right_son(node));
	avl_tree_node_update_height(node);

	avl_tree_set_root(l_tree, NULL);
	avl_tree_set_root(r_tree, NULL);
	avl_tree_set_root(l_tree_l, NULL);
	avl_tree_set_root(l_tree_r, NULL);
	avl_tree_destroy(l_tree);
	avl_tree_destroy(r_tree);
	avl_tree_destroy(l_tree_l);
	avl_tree_destroy(l_tree_r);
}

static void left_rotation(const avl_tree_t handle, const avl_tree_node_t node) {
	struct avl_tree* tree = (struct avl_tree*)handle;

	avl_tree_node_t right_son = avl_tree_node_get_right_son(node);
	avl_tree_node_swap(node, right_son);

	avl_tree_t r_tree = avl_tree_cut_right(tree, node);
	avl_tree_t l_tree = avl_tree_cut_left(tree, node);
	avl_tree_t r_tree_l = avl_tree_cut_left(r_tree, right_son);
	avl_tree_t r_tree_r = avl_tree_cut_right(r_tree, right_son);

	avl_tree_insert_as_left_subtree(r_tree, avl_tree_get_root(r_tree), l_tree);
	avl_tree_insert_as_right_subtree(r_tree, avl_tree_get_root(r_tree), r_tree_l);
	avl_tree_insert_as_left_subtree(tree, node, r_tree);
	avl_tree_insert_as_right_subtree(tree, node, r_tree_r);

	avl_tree_node_update_height(avl_tree_node_get_left_son(node));
	avl_tree_node_update_height(node);

	avl_tree_set_root(r_tree, NULL);
	avl_tree_set_root(l_tree, NULL);
	avl_tree_set_root(r_tree_l, NULL);
	avl_tree_set_root(r_tree_r, NULL);
	avl_tree_destroy(r_tree);
	avl_tree_destroy(l_tree);
	avl_tree_destroy(r_tree_l);
	avl_tree_destroy(r_tree_r);
}

static void rotate(const avl_tree_t handle, const avl_tree_node_t node) {
	struct avl_tree* tree = (struct avl_tree*)handle;

	long balanced_factor = avl_tree_node_get_balance_factor(node);
	if (balanced_factor == 2) {	//height of the node's left son is 2 times greater
than the right one
		if (avl_tree_node_get_balance_factor(avl_tree_node_get_left_son(node)) >= 0) {
	//LL balancing
			right_rotation(tree, node);
		}
		else { //LR balancing
			left_rotation(tree, avl_tree_node_get_left_son(node));
			right_rotation(tree, node);
		}
	}
	else if (balanced_factor == -2) { //height of the node's right son is 2 times greater
than the left one
		if (avl_tree_node_get_balance_factor(avl_tree_node_get_left_son(node)) >= 0) {
	//RR balancing
			left_rotation(tree, node);
		}
		else { //RL balancing
			right_rotation(tree, avl_tree_node_get_right_son(node));
			left_rotation(tree, node);
		}
	}
}

static void balance_insert(const avl_tree_t handle, const avl_tree_node_t node) {
```

```c
	struct avl_tree* tree = (struct avl_tree*)handle;

	avl_tree_node_t current = avl_tree_node_get_father(node);
	while (current) {
		if (abs(avl_tree_node_get_balance_factor(current)) >= 2) {
			break;
		}
		else {
			avl_tree_node_update_height(current);
			current = avl_tree_node_get_father(current);
		}
	}
	if (current) {
		rotate(tree, current);
	}
}

static void balance_delete(const avl_tree_t handle, const avl_tree_node_t node) {
	struct avl_tree* tree = (struct avl_tree*)handle;

	avl_tree_node_t current = avl_tree_node_get_father(node);
	while (current) {
		if (abs(avl_tree_node_get_balance_factor(current)) == 2) {
			rotate(tree, current);
		}
		else {
			avl_tree_node_update_height(current);
		}
		current = avl_tree_node_get_father(current);
	}
}

static int cut_single_son(const avl_tree_t handle, const avl_tree_node_t node) {
	struct avl_tree* tree = (struct avl_tree*)handle;
	struct avl_tree* cutted = (struct avl_tree*)handle;
	cutted = avl_tree_cut_one_son_node(tree, node);
	balance_delete(tree, node);
	return avl_tree_destroy(cutted);
}

static int subtree_nodes_number(const avl_tree_t node) {
	int nodes_number = 0;
	_stack_t stack = stack_init();
	if (node) {
		stack_push(stack, node);
		while (!stack_is_empty(stack)) {
			struct binary_tree_node* current_node;
			current_node = stack_pop(stack);
			nodes_number++;
			if (avl_tree_node_get_left_son(current_node)) {
				stack_push(stack, avl_tree_node_get_left_son(current_node));
			}
			if (avl_tree_node_get_right_son(current_node)) {
				stack_push(stack, avl_tree_node_get_right_son(current_node));
			}
		}
	}
	stack_destroy(stack);
	return nodes_number;
}

static int defualt_comparison_function(const void* key1, const void* key2) {
	if (key1 < key2) {
		return -1;
```

```
        }
        else if (key1 > key2) {
                return 1;
        }
        return 0;
}
```

avl_tree_node.h

```c
#pragma once

typedef void* avl_tree_node_t;

avl_tree_node_t avl_tree_node_init(const void* key, const void* value);

int avl_tree_node_destroy(const avl_tree_node_t handle);

long avl_tree_node_degree(const avl_tree_node_t handle);

int avl_tree_node_swap(const avl_tree_node_t handle1, const avl_tree_node_t handle2);

void* avl_tree_node_get_key(const avl_tree_node_t handle);

void* avl_tree_node_get_value(const avl_tree_node_t handle);

int avl_tree_node_get_height(const avl_tree_node_t handle);

int avl_tree_node_set_height(const avl_tree_node_t handle, const int height);

void avl_tree_node_update_height(const avl_tree_node_t handle);

int avl_tree_node_get_balance_factor(const avl_tree_node_t handle);

avl_tree_node_t avl_tree_node_get_father(const avl_tree_node_t handle);

int avl_tree_node_set_father(const avl_tree_node_t handle, const avl_tree_node_t father);

avl_tree_node_t avl_tree_node_get_left_son(const avl_tree_node_t handle);

int avl_tree_node_set_left_son(const avl_tree_node_t handle, const avl_tree_node_t
left_son);

avl_tree_node_t avl_tree_node_get_right_son(const avl_tree_node_t handle);

int avl_tree_node_set_right_son(const avl_tree_node_t handle, const avl_tree_node_t
right_son);

int avl_tree_node_is_left_son(const avl_tree_node_t handle);

int avl_tree_node_is_right_son(const avl_tree_node_t handle);

avl_tree_node_t avl_tree_node_get_max(const avl_tree_node_t node);

avl_tree_node_t avl_tree_node_get_pred(const avl_tree_node_t node);
```

avl_tree_node.c

```c
#include "avl_tree_node.h"

#include <stdlib.h>

#ifndef max
#define max(a,b) (((a) > (b)) ? (a) : (b))
#endif

struct avl_tree_node {
	int height;
	void* key;
	void* value;
	avl_tree_node_t father;
	avl_tree_node_t left_son;
	avl_tree_node_t right_son;
};

avl_tree_node_t avl_tree_node_init(const void* key, const void* value) {
	struct avl_tree_node* node;
	if ((node = malloc(sizeof(struct avl_tree_node))) == NULL) {
		return NULL;
	}
	node->height = 0;
	node->key = (void*)key;
	node->value = (void*)value;
	node->father = NULL;
	node->left_son = NULL;
	node->right_son = NULL;
	return node;
}

int avl_tree_node_destroy(const avl_tree_node_t handle) {
	struct avl_tree_node* node = (struct avl_tree_node*)handle;
	free(node);
	return 0;
}

inline long avl_tree_node_degree(const avl_tree_node_t handle) {
	struct avl_tree_node* node = (struct avl_tree_node*)handle;
	long degree = 0;

	if (node) {
		if (node->left_son) {
			degree++;
		}
		if (node->right_son) {
			degree++;
		}
	}
	return degree;
}

inline int avl_tree_node_swap(const avl_tree_node_t handle1, const avl_tree_node_t handle2)
{
	struct avl_tree_node* node1 = (struct avl_tree_node*)handle1;
	struct avl_tree_node* node2 = (struct avl_tree_node*)handle2;
	void* tmp;
	tmp = node1->key;
	node1->key = node2->key;
	node2->key = tmp;
	tmp = node1->value;
	node1->value = node2->value;
	node2->value = tmp;
	return 0;
}

inline void* avl_tree_node_get_key(const avl_tree_node_t handle) {
	struct avl_tree_node* node = (struct avl_tree_node*)handle;
	return node ? node->key : NULL;
}

inline void* avl_tree_node_get_value(const avl_tree_node_t handle) {
	struct avl_tree_node* node = (struct avl_tree_node*)handle;
	return node ? node->value : NULL;
}

inline int avl_tree_node_get_height(const avl_tree_node_t handle) {
	struct avl_tree_node* node = (struct avl_tree_node*)handle;
	return node ? node->height : -1;
}

int avl_tree_node_set_height(const avl_tree_node_t handle, int height) {
	struct avl_tree_node* node = (struct avl_tree_node*)handle;
	if (node) {
		node->height = height;
		return 0;
	}
	return 1;
}

void avl_tree_node_update_height(const avl_tree_node_t handle) {
	struct avl_tree_node* node = (struct avl_tree_node*)handle;
	if (node) {
		int left_son_height = 
avl_tree_node_get_height(avl_tree_node_get_left_son(node));
		int right_son_height = 
avl_tree_node_get_height(avl_tree_node_get_right_son(node));
		avl_tree_node_set_height(node, max(left_son_height, right_son_height) + 1);
	}
}

int avl_tree_node_get_balance_factor(const avl_tree_node_t handle) {
	struct avl_tree_node* node = (struct avl_tree_node*)handle;
	if (node) {
		int left_son_height = 
avl_tree_node_get_height(avl_tree_node_get_left_son(node));
		int right_son_height = 
avl_tree_node_get_height(avl_tree_node_get_right_son(node));
		return  left_son_height - right_son_height;
	}
	return 0;
}

inline avl_tree_node_t avl_tree_node_get_father(const avl_tree_node_t handle) {
	struct avl_tree_node* node = (struct avl_tree_node*)handle;
	return node ? node->father : NULL;
}

int avl_tree_node_set_father(const avl_tree_node_t handle, const avl_tree_node_t father) {
	struct avl_tree_node* node = (struct avl_tree_node*)handle;
	if (node) {
		node->father = father;
		return 0;
	}
	return 1;
}
```

```c
inline avl_tree_node_t avl_tree_node_get_left_son(const avl_tree_node_t handle) {
	struct avl_tree_node* node = (struct avl_tree_node*)handle;
	return node ? node->left_son : NULL;
}

int avl_tree_node_set_left_son(const avl_tree_node_t handle, const avl_tree_node_t left_son)
{
	struct avl_tree_node* node = (struct avl_tree_node*)handle;
	if (node) {
		node->left_son = left_son;
		return 0;
	}
	return 1;
}

inline avl_tree_node_t avl_tree_node_get_right_son(const avl_tree_node_t handle) {
	struct avl_tree_node* node = (struct avl_tree_node*)handle;
	return node ? node->right_son : NULL;
}

int avl_tree_node_set_right_son(const avl_tree_node_t handle, const avl_tree_node_t
right_son) {
	struct avl_tree_node* node = (struct avl_tree_node*)handle;
	if (node) {
		node->right_son = right_son;
		return 0;
	}
	return 1;
}

int avl_tree_node_is_left_son(const avl_tree_node_t handle) {
	struct avl_tree_node* node = (struct avl_tree_node*)handle;
	if (node) {
		struct avl_tree_node* father = node->father;
		if (father) {
			if (father->left_son == node) {
				return 1;
			}
		}
	}
	return 0;
}

int avl_tree_node_is_right_son(const avl_tree_node_t handle) {
	struct avl_tree_node* node = (struct avl_tree_node*)handle;
	if (node) {
		struct avl_tree_node* father = node->father;
		if (father) {
			if (father->right_son == node) {
				return 1;
			}
		}
	}
	return 0;
}

inline avl_tree_node_t avl_tree_node_get_max(const avl_tree_node_t handle) {
	struct avl_tree_node* node = (struct avl_tree_node*)handle;
	while (avl_tree_node_get_right_son(node)) {
		node = avl_tree_node_get_right_son(node);
	}
	return node;
}
```

```c
inline avl_tree_node_t avl_tree_node_get_pred(const avl_tree_node_t handle) {
	struct avl_tree_node* node = (struct avl_tree_node*)handle;
	if (node == NULL) {
		return NULL;
	}
	if (avl_tree_node_get_left_son(node)) {
		return avl_tree_node_get_max(avl_tree_node_get_left_son(node));
	}
	while (avl_tree_node_is_left_son(node)) {
		node = avl_tree_node_get_father(node);
	}
	return node;
}
```

flag.h

```c
#pragma once

typedef void* flag_t;

flag_t flag_init();

void flag_destroy(const flag_t handle);

int flag_status(const flag_t handle);

void flag_set(const flag_t handle);

void flag_unset(const flag_t handle);
```

flag.c

```c
#include "flag.h"
#include <stdlib.h>

#define byte unsigned char

struct flag {
	byte value;
};

flag_t flag_init() {
	struct flag* flag;
	if ((flag = malloc(sizeof(struct flag))) == NULL) {
		return NULL;
	}
	flag_unset(flag);
	return flag;
}

void flag_destroy(const flag_t handle) {
	struct flag* flag = (struct flag*)handle;
	free(flag);
}

int flag_status(const flag_t handle) {
	struct flag* flag = (struct flag*)handle;
	return flag->value;
}

void flag_set(const flag_t handle) {
	struct flag* flag = (struct flag*)handle;
	flag->value = 1;
}

void flag_unset(const flag_t handle) {
	struct flag* flag = (struct flag*)handle;
	flag->value = 0;
}
```

concurrent_flag.h

```c
#pragma once

typedef void* concurrent_flag_t;

concurrent_flag_t concurrent_flag_init();

int concurrent_flag_destroy(const concurrent_flag_t handle);

int concurrent_flag_status(const concurrent_flag_t handle, int* result);

int concurrent_flag_set(const concurrent_flag_t handle);

int concurrent_flag_unset(const concurrent_flag_t handle);
```

concurrent_flag.c

```c
#include "concurrent_flag.h"
#include <stdlib.h>
#include <pthread.h>
#include <errno.h>

#include "flag.h"

struct concurrent_flag {
	flag_t flag;
	pthread_mutex_t mutex;
};

concurrent_flag_t concurrent_flag_init() {
	int ret;
	struct concurrent_flag* concurrent_flag;
	if ((concurrent_flag = malloc(sizeof(struct concurrent_flag))) == NULL) {
		return NULL;
	}
	if ((concurrent_flag->flag = flag_init()) == NULL) {
		free(concurrent_flag);
		return NULL;
	}
	while ((ret = pthread_mutex_init(&concurrent_flag->mutex, NULL)) && errno == EINTR);
	if (ret) {
		flag_destroy(concurrent_flag->flag);
		free(concurrent_flag);
		return NULL;
	}
	return concurrent_flag;
}

int concurrent_flag_destroy(const concurrent_flag_t handle) {
	int ret;
	struct concurrent_flag* concurrent_flag = (struct concurrent_flag*)handle;
	while ((ret = pthread_mutex_destroy(&concurrent_flag->mutex)) && errno == EINTR);
	if (ret) {
		return 1;
	}
	flag_destroy(concurrent_flag->flag);
	free(concurrent_flag);
	return 0;
}

int concurrent_flag_status(const concurrent_flag_t handle, int* result) {
	int ret;
	struct concurrent_flag* concurrent_flag = (struct concurrent_flag*)handle;
	while ((ret = pthread_mutex_lock(&concurrent_flag->mutex)) && errno == EINTR);
	if (ret) {
		return 1;
	}
	*result = flag_status(concurrent_flag->flag);
	while ((ret = pthread_mutex_unlock(&concurrent_flag->mutex)) && errno == EINTR);
	if (ret) {
		return 1;
	}
	return 0;
}

int concurrent_flag_set(const concurrent_flag_t handle) {
	int ret;
	struct concurrent_flag* concurrent_flag = (struct concurrent_flag*)handle;
	while ((ret = pthread_mutex_lock(&concurrent_flag->mutex)) && errno == EINTR);
	if (ret) {
		return 1;
	}
	flag_set(concurrent_flag->flag);
	while ((ret = pthread_mutex_unlock(&concurrent_flag->mutex)) && errno == EINTR);
	if (ret) {
		return 1;
	}
	return 0;
}

int concurrent_flag_unset(const concurrent_flag_t handle) {
	int ret;
	struct concurrent_flag* concurrent_flag = (struct concurrent_flag*)handle;
	while ((ret = pthread_mutex_lock(&concurrent_flag->mutex)) && errno == EINTR);
	if (ret) {
		return 1;
	}
	flag_unset(concurrent_flag->flag);
	while ((ret = pthread_mutex_unlock(&concurrent_flag->mutex)) && errno == EINTR);
	if (ret) {
		return 1;
	}
	return 0;
}
```

stack.h

```
#pragma once

typedef void* _stack_t;

_stack_t stack_init();

void stack_destroy(const _stack_t handle);

int stack_is_empty(const _stack_t handle);

int stack_push(const _stack_t handle, void* item);

void* stack_pop(const _stack_t handle);

void* stack_peek(const _stack_t handle);
```

stack.c

```c
#include "stack.h"
#include <stdlib.h>

struct stack_node {
	void* data;
	struct stack_node* next;
};

_stack_t stack_init() {
	struct stack_node* stack;
	if ((stack = malloc(sizeof(struct stack_node))) == NULL) {
		return NULL;
	}
	stack->data = NULL;
	stack->next = NULL;
	return stack;
}

void stack_destroy(const _stack_t handle) {
	struct stack_node* stack = (struct stack_node*)handle;
	while (!stack_is_empty(stack)) {
		stack_pop(stack);
	}
	free(stack);
}

int stack_is_empty(const _stack_t handle) {
	struct stack_node* stack = (struct stack_node*)handle;
	return !stack->next;
}

int stack_push(const _stack_t handle, void* item) {
	struct stack_node* stack = (struct stack_node*)handle;
	struct stack_node* node;
	if ((node = malloc(sizeof(struct stack_node))) == NULL) {
		return 1;
	}
	node->data = stack->data;
	node->next = stack->next;
	stack->data = item;
	stack->next = node;
	return 0;
}

void* stack_pop(const _stack_t handle) {
	struct stack_node* stack = (struct stack_node*)handle;
	struct stack_node* tmp = stack->next;
	void* popped = stack->data;
	if (!stack_is_empty(stack)) {
		stack->data = stack->next->data;
		stack->next = stack->next->next;
		free(tmp);
	}
	return popped;
}

void* stack_peek(const _stack_t handle) {
	struct stack_node* stack = (struct stack_node*)handle;
	return stack->data;
}
```

queue.h

```c
#pragma once

typedef void* queue_t;

queue_t queue_init();

void queue_destroy(const queue_t handle);

int queue_is_empty(const queue_t handle);

int queue_enqueue(const queue_t handle, void* item);

void* queue_dequeue(const queue_t handle);
```

queue.c

```c
#include "queue.h"
#include <stdlib.h>

#include "stack.h"

struct queue {
	_stack_t stack_in;
	_stack_t stack_out;
};

queue_t queue_init() {
	struct queue* queue;
	if ((queue = malloc(sizeof(struct queue))) == NULL) {
		return NULL;
	}
	if ((queue->stack_in = stack_init()) == NULL) {
		free(queue);
		return NULL;
	}
	if ((queue->stack_out = stack_init()) == NULL) {
		free(queue);
		return NULL;
	}
	return queue;
}

void queue_destroy(const queue_t handle) {
	struct queue* queue = (struct queue*)handle;
	stack_destroy(queue->stack_in);
	stack_destroy(queue->stack_out);
	free(queue);
}

int queue_is_empty(const queue_t handle) {
	struct queue* queue = (struct queue*)handle;
	return (stack_is_empty(queue->stack_in) && stack_is_empty(queue->stack_out));
}

int queue_enqueue(const queue_t handle, void* item) {
	struct queue* queue = (struct queue*)handle;
	return stack_push(queue->stack_in, item);
}

void* queue_dequeue(const queue_t handle) {
	struct queue* queue = (struct queue*)handle;
	if (stack_is_empty(queue->stack_out)) {
		while (!stack_is_empty(queue->stack_in)) {
			stack_push(queue->stack_out, stack_pop(queue->stack_in));
		}
	}
	return stack_pop(queue->stack_out);
}
```

concurrent_queue.h

```c
#pragma once

typedef void* concurrent_queue_t;

concurrent_queue_t concurrent_queue_init();

int concurrent_queue_destroy(const concurrent_queue_t handle);

int concurrent_queue_is_empty(const concurrent_queue_t handle, int* result);

int concurrent_queue_enqueue(const concurrent_queue_t handle, void* item);

int concurrent_queue_dequeue(const concurrent_queue_t handle, void** result);
```

concurrent_queue.c

```
#include "concurrent_queue.h"
#include <stdlib.h>
#include <pthread.h>
#include <errno.h>

#include "queue.h"

struct concurrent_queue {
	queue_t queue;
	pthread_mutex_t mutex;
};

concurrent_queue_t concurrent_queue_init() {
	int ret;
	struct concurrent_queue* concurrent_queue;
	if ((concurrent_queue = malloc(sizeof(struct concurrent_queue))) == NULL) {
		return NULL;
	}
	if ((concurrent_queue->queue = queue_init()) == NULL) {
		free(concurrent_queue);
		return NULL;
	}
	while ((ret = pthread_mutex_init(&concurrent_queue->mutex, NULL)) && errno == EINTR);
	if (ret) {
		queue_destroy(concurrent_queue->queue);
		free(concurrent_queue);
		return NULL;
	}
	return concurrent_queue;
}

int concurrent_queue_destroy(const concurrent_queue_t handle) {
	int ret;
	struct concurrent_queue* concurrent_queue = (struct concurrent_queue*)handle;
	while ((ret = pthread_mutex_destroy(&concurrent_queue->mutex)) && errno == EINTR);
	if (ret) {
		return 1;
	}
	queue_destroy(concurrent_queue->queue);
	free(concurrent_queue);
	return 0;
}

int concurrent_queue_is_empty(const concurrent_queue_t handle, int* result) {
	int ret;
	struct concurrent_queue* concurrent_queue = (struct concurrent_queue*)handle;
	while ((ret = pthread_mutex_lock(&concurrent_queue->mutex)) && errno == EINTR);
	if (ret) {
		return 1;
	}
	*result = queue_is_empty(concurrent_queue->queue);
	while ((ret = pthread_mutex_unlock(&concurrent_queue->mutex)) && errno == EINTR);
	if (ret) {
		return 1;
	}
	return 0;
}

int concurrent_queue_enqueue(const concurrent_queue_t handle, void* item) {
	int ret;
	int result;
	struct concurrent_queue* concurrent_queue = (struct concurrent_queue*)handle;
```

```c
	while ((ret = pthread_mutex_lock(&concurrent_queue->mutex)) && errno == EINTR);
	if (ret) {
		return 1;
	}
	result = queue_enqueue(concurrent_queue->queue, item);
	while ((ret = pthread_mutex_unlock(&concurrent_queue->mutex)) && errno == EINTR);
	if (ret) {
		return 1;
	}
	return result;
}

int concurrent_queue_dequeue(const concurrent_queue_t handle, void** result) {
	int ret;
	struct concurrent_queue* concurrent_queue = (struct concurrent_queue*)handle;
	while ((ret = pthread_mutex_lock(&concurrent_queue->mutex)) && errno == EINTR);
	if (ret) {
		return 1;
	}
	*result = queue_dequeue(concurrent_queue->queue);
	while ((ret = pthread_mutex_unlock(&concurrent_queue->mutex)) && errno == EINTR);
	if (ret) {
		return 1;
	}
	return 0;
}
```