Università degli studi di Napoli Parthenope

Laboratorio di Reti di Calcolatori

Anno 2023/2024

Progetto Università

Nicola Menale 0124/002551 Giulio Simpatico 0124/002509

Professore Emanuel Di Nardo

Panoramica del Progetto

Il progetto è un sistema distribuito basato su architettura client-server a tre livelli, progettato per gestire le operazioni di visualizzazione e prenotazione degli esami universitari. Il sistema coinvolge tre entità principali: il **Client Studente**, il **Server Segreteria**, e il **Server Universitario**. Queste componenti collaborano per fornire agli studenti un'interfaccia funzionale per interagire con la segreteria universitaria e gestire le prenotazioni degli esami.

Obiettivo Principale

L'obiettivo del sistema è consentire agli studenti di visualizzare gli esami disponibili e di effettuare prenotazioni, mentre i server intermedi gestiscono e inoltrano le richieste e risposte tra i vari componenti del sistema.

Componenti del Sistema

1. Client Studente

- Funzione: L'interfaccia utilizzata dallo studente per inviare richieste di visualizzazione e prenotazione esami.
- Modalità di Operazione: Il client invia richieste al Server Segreteria per visualizzare gli esami disponibili o per prenotare un esame specifico. Riceve poi le risposte inoltrate dal Server Segreteria.

2. Server Segreteria

- Funzione: Il server intermediario che gestisce le comunicazioni tra il Client Studente e il Server Universitario.
- Modalità di Operazione: Il Server Segreteria riceve le richieste dal Client Studente e le inoltra al Server Universitario. Inoltre, invia eventuali richieste di creazione di esami. Una volta ricevuta la risposta dal Server Universitario, la inoltra al Client Studente.

3. Server Universitario

- Funzione: Il server centrale che gestisce le informazioni sugli esami e le prenotazioni.
- Modalità di Operazione: Il Server Universitario riceve richieste dal Server Segreteria, le elabora, e invia le risposte indietro al Server Segreteria. È responsabile di mantenere i dati sugli esami disponibili e sulle prenotazioni effettuate.

Flusso di Lavoro del Sistema

1. Richiesta da parte del Client Studente

- Lo studente utilizza il **Client Studente** per inviare una richiesta, come visualizzare gli esami disponibili o prenotarsi per un esame.

2. Inoltro della Richiesta

- La richiesta del Client Studente viene ricevuta dal **Server Segreteria**, che agisce come intermediario. Il Server Segreteria inoltra questa richiesta al **Server Universitario**.

3. Elaborazione della Richiesta

- Il **Server Universitario** riceve la richiesta dal Server Segreteria, la elabora (ad esempio, cercando gli esami disponibili o effettuando una prenotazione) e genera una risposta appropriata.

4. Inoltro della Risposta

- La risposta del Server Universitario viene inviata al **Server Segreteria**, che la inoltra al **Client Studente**.

5. Risposta al Client Studente

- Il Client Studente riceve la risposta finale, che può essere la lista degli esami disponibili o la conferma di una prenotazione effettuata.

Caratteristiche Chiave

- **Architettura a Tre Livelli**: Il sistema utilizza un'architettura a tre livelli, in cui il Client Studente, il Server Segreteria, e il Server Universitario sono distinti e comunicano tra loro. Questo approccio migliora la modularità e la scalabilità del sistema.

- **Intermediazione delle Richieste**: Il Server Segreteria funge da intermediario, semplificando le comunicazioni tra il Client Studente e il Server Universitario e permettendo una gestione centralizzata delle richieste e delle risposte.

- **Multitasking e Concorrenza**: Sia il Server Segreteria che il Server Universitario sono progettati per gestire più connessioni contemporaneamente, garantendo che più studenti possano accedere al sistema in parallelo senza problemi di prestazioni.

Requisiti del Progetto

Il progetto definisce una serie di requisiti necessari per il corretto funzionamento del sistema distribuito, suddivisi tra requisiti funzionali e requisiti di sistema. Questi requisiti coprono le funzionalità principali e le necessità software e hardware per i tre componenti principali: il **Client Studente**, il **Server Segreteria**, e il **Server Universitario**.

Requisiti Funzionali

Questi requisiti definiscono le funzionalità essenziali che il sistema deve offrire:

- **Autenticazione e Identificazione dello Studente**: Ogni studente deve essere identificato tramite un ID univoco generato casualmente all'avvio del client. Non è prevista una fase di login tradizionale, ma ogni richiesta sarà associata a un ID studente.

- **Visualizzazione degli Esami Disponibili**: Il sistema deve permettere allo studente di visualizzare una lista di esami disponibili, comprensiva del nome del corso e della data dell'esame.

- **Prenotazione Esame**: Il sistema deve consentire agli studenti di prenotare un esame. Una volta effettuata la prenotazione, lo studente deve ricevere una conferma di successo o un messaggio di errore in caso di problemi (es. esame già prenotato o non disponibile).

- **Creazione di Esami**: Il Server Segreteria deve poter inviare richieste al Server Universitario per creare nuove date d'esame.

- **Inoltro delle Richieste e Risposte**: Il Server Segreteria deve inoltrare correttamente le richieste ricevute dal Client Studente al Server Universitario e viceversa per le risposte.

- **Gestione degli Errori**: In caso di errori (es. connessione interrotta, richiesta non valida), il sistema deve essere in grado di gestirli in modo appropriato, notificando l'utente e permettendo il recupero o la chiusura dell'applicazione in modo sicuro.

Requisiti di Sistema

Questi requisiti definiscono l'ambiente hardware e software necessario per eseguire il sistema:

Sistema Operativo:

- Il sistema deve essere eseguibile su sistemi operativi Unix-like (es. Linux, macOS).

Librerie e Dipendenze:

- Il sistema utilizza le librerie standard del C, come `<sys/types.h>`, `<unistd.h>`, `<arpa/inet.h>`, `<sys/socket.h>`, `<stdio.h>`, `<errno.h>`, `<string.h>`, `<stdlib.h>`, e `<time.h>`.

- Non sono necessarie librerie esterne o dipendenze non standard.

Rete:

- Una rete TCP/IP funzionante è necessaria per la comunicazione tra il Client Studente, il Server Segreteria, e il Server Universitario. Le porte utilizzate devono essere aperte e accessibili tra i diversi componenti del sistema.

Compilatore:

- Il codice è scritto in linguaggio C e deve essere compilato con un compilatore C standard, come GCC. La compatibilità è garantita con le versioni moderne di GCC (es. GCC 7.5 e successivi).

Considerazioni di Implementazione

- Gestione della Concorrenza: L'uso di `select()` nel codice garantisce che il sistema possa gestire più connessioni simultanee, un requisito fondamentale per il corretto funzionamento in ambienti multiutente.

- Gestione degli Errori e Debugging: È importante che tutte le operazioni di lettura e scrittura sui socket siano gestite con cura, per evitare deadlock o perdita di dati. Il codice deve includere controlli robusti per gestire eventuali errori durante la comunicazione.

Struttura del Codice

Il progetto è composto da tre principali componenti software, ciascuno implementato in un file sorgente C separato:

1. **Client Studente** (`Studente.c`)
2. **Server Segreteria** (`Server_Segreteria.c`)
3. **Server Universitario** (`Server_Universitario.c`)

Ciascun componente è progettato per svolgere specifiche funzioni all'interno del sistema, e la loro implementazione è suddivisa in moduli e funzioni per mantenere il codice organizzato e leggibile. Di seguito è illustrata la struttura del codice per ogni componente.

Client Studente (`Studente.c`)

Il client studente è progettato per permettere agli studenti di interagire con il sistema di gestione degli esami, consentendo loro di visualizzare le date degli esami disponibili e di effettuare prenotazioni. Il client si connette al server tramite un socket TCP e invia richieste per ottenere informazioni o effettuare prenotazioni. Il codice è strutturato per garantire una comunicazione affidabile e gestire input utente in modo efficiente.

Struttura Generale del Client

Il client utilizza un singolo socket TCP per comunicare con il server. La gestione dell'interazione utente avviene tramite un menu che permette di selezionare diverse opzioni, come visualizzare gli esami disponibili o richiedere una prenotazione.

Funzione `main`

La funzione `main` è il punto di ingresso del programma e si occupa della configurazione e gestione della connessione al server. Di seguito i passaggi principali:

- **Inizializzazione del generatore di numeri casuali**: Viene generato un ID studente casuale per simulare l'identificazione dello studente.
- **Configurazione del socket**: Viene creato un socket TCP che si connette all'indirizzo IP del server passato come argomento della riga di comando.

- **Connessione al server**: Il client stabilisce una connessione con il server tramite il socket configurato.
- **Gestione delle operazioni dello studente**: La funzione `Student_Function` è chiamata per gestire il menu delle operazioni che lo studente può eseguire, come visualizzare gli esami o prenotarne uno.
- **Chiusura della connessione**: Alla fine della sessione, il socket viene chiuso correttamente.

Funzione `Student_Function`

Questa funzione gestisce l'interazione tra lo studente e il server. Un menu permette allo studente di scegliere tra diverse opzioni. La funzione utilizza `select` per gestire contemporaneamente input da tastiera e dati ricevuti dal server.

- **Visualizzazione del menu**: Mostra le opzioni disponibili, come visualizzare gli esami o richiedere una prenotazione.
- **Gestione dell'input utente**: L'input utente è gestito in modo da validare le scelte e chiamare le funzioni appropriate (`Esami_Disponibili` o `Richiesta_Prenotazione`).
- **Gestione del socket**: Utilizza `select` per verificare se ci sono dati disponibili dal server o input dall'utente, permettendo una gestione non bloccante della comunicazione.

Funzione `Esami_Disponibili`

Questa funzione permette allo studente di visualizzare la lista degli esami disponibili e di richiedere maggiori informazioni su di essi.

- **Mostra il menu degli esami**: Presenta un elenco di esami disponibili per cui lo studente può richiedere informazioni.
- **Invio della richiesta al server**: A seconda dell'esame scelto, il client invia un messaggio al server richiedendo informazioni sull'esame.
- **Ricezione e visualizzazione della risposta**: Il client riceve la lista degli esami disponibili dal server e la stampa a schermo.

Funzione `Richiesta_Prenotazione`

Questa funzione gestisce la prenotazione di un esame da parte dello studente.

- **Richiesta della lista degli esami**: Il client invia una richiesta al server per ottenere la lista degli esami disponibili.
- **Visualizzazione della lista e selezione dell'esame**: Lo studente seleziona un esame dalla lista ricevuta.
- **Invio della richiesta di prenotazione**: Il client invia la richiesta di prenotazione al server, includendo l'ID dello studente e l'esame scelto.
- **Ricezione della conferma**: Il client riceve una conferma dal server riguardante lo stato della prenotazione e la visualizza a schermo.

Funzione `FullWrite`

La funzione `FullWrite` è utilizzata per garantire che tutti i dati siano inviati correttamente sul socket, gestendo eventuali interruzioni di sistema (`EINTR`).

- **Scrittura affidabile**: La funzione si assicura che l'intero buffer venga scritto, riprovando in caso di interruzione.
- **Gestione degli errori**: Se si verifica un errore diverso da `EINTR`, la funzione interrompe l'operazione e segnala il fallimento.

Server Segreteria (`Segreteria.c`)

Il server della segreteria è responsabile della gestione delle date d'esame per gli studenti e funge anche da intermediario tra i client studenti e il server universitario. Il server accetta richieste dai client e interagisce con un server universitario per l'inserimento e la verifica delle date d'esame. Il codice è strutturato per gestire più client contemporaneamente utilizzando il multithreading. Di seguito è presentata la descrizione dettagliata delle funzioni principali che costituiscono il server della segreteria.

Struttura Generale del Server

Il server utilizza un socket TCP per accettare connessioni dai client e un altro socket per comunicare con il server universitario. Il server è in ascolto su una porta specifica e avvia un thread per ogni client connesso, permettendo la gestione concorrente delle richieste.

Funzione `main`

La funzione `main` è il punto di ingresso del programma. Essa configura il server per accettare connessioni client e gestisce l'inserimento degli esami tramite un'interfaccia terminale.

- **Configurazione del socket del server**: Viene creato un socket TCP e associato a un indirizzo IP e una porta specifica tramite le funzioni `socket`, `bind` e `listen`.
- **Thread per la gestione degli esami**: Viene creato un thread separato per la gestione dell'inserimento degli esami attraverso l'interfaccia terminale (`manage_exams`).
- **Ciclo di accettazione dei client**: Il server rimane in ascolto di nuove connessioni. Per ogni client connesso, viene creato un thread separato che esegue la funzione `handle_client`.
- **Chiusura del server**: Quando il server deve terminare, viene chiuso il socket del server e i thread vengono terminati.

Gestione Client con `handle_client`

La funzione `handle_client` gestisce la comunicazione con un singolo client. Ogni volta che un client si connette, il server crea un nuovo thread che esegue questa funzione.

- **Creazione e connessione del socket universitario**: Se non esiste una connessione attiva con il server universitario, viene creato un nuovo socket e si tenta di stabilire la connessione.

- **Comunicazione con il client**: Il server legge i dati inviati dal client e li inoltra al server universitario. La risposta dal server universitario viene quindi inviata al client.
- **Chiusura della connessione**: Alla fine della comunicazione, il socket del client viene chiuso.

Funzione `Inserisci_esame`

Questa funzione gestisce l'inserimento degli esami attraverso un'interfaccia utente terminale. L'utente può selezionare un esame da una lista e specificare la data dell'esame.

- **Selezione dell'esame**: L'utente seleziona un esame tra quelli disponibili.
- **Inserimento della data**: L'utente inserisce il giorno, il mese e l'anno dell'esame. La data viene quindi validata e normalizzata (ad esempio, "1" diventa "01").
- **Invio della data**: La data viene formattata e inviata al server universitario. Se l'invio ha successo, il server conferma l'aggiunta dell'esame.

Validazione e Formattazione della Data

La validazione della data è gestita da diverse funzioni:
- `**is_digit_string**`: Verifica che una stringa contenga solo cifre.
- `**is_valid_day**`: Controlla che il giorno, il mese e l'anno siano validi considerando mesi e anni bisestili.
- `**normalize_date_parts**`: Normalizza giorno e mese aggiungendo uno zero iniziale se necessario.
- `**validate_date**`: Combina le precedenti funzioni per verificare se una data è valida nel formato `dd-mm-yyyy`.

La funzione `**process_date**` normalizza e valida la data, restituendo una stringa formattata nel caso in cui la data sia valida.

Scrittura Completa con `FullWrite`

La funzione `**FullWrite**` gestisce l'invio di dati su un socket assicurandosi che l'intero buffer sia scritto. Gestisce inoltre l'interruzione di sistema `EINTR`.

Server Universitario (`Server_Universitario.c`)

Il server universitario è progettato per gestire le richieste relative alle prenotazioni e alle date degli esami. Questo server si interfaccia direttamente con i client studenti, accettando richieste di visualizzazione degli esami disponibili, prenotazioni degli esami e l'aggiunta di nuovi esami. Il codice è strutturato per gestire multiple connessioni contemporaneamente e garantire la correttezza delle operazioni tramite l'uso di meccanismi di gestione della comunicazione.

Struttura Generale del Server

Il server universitario utilizza un socket TCP per accettare connessioni dai client, gestendo le richieste in modo sequenziale e sincronizzato. Il server rimane in ascolto su una porta specifica, accettando connessioni dai client e rispondendo in base alle richieste ricevute. È possibile gestire esami predefiniti e nuove aggiunte, garantendo che le operazioni di prenotazione siano eseguite correttamente e che non ci siano duplicazioni.

Funzione main

La funzione `main` è il punto di ingresso del programma e si occupa della configurazione e gestione del server.

- **Configurazione del Socket del Server**: Il socket TCP è creato e associato a un indirizzo IP e una porta specifica attraverso le funzioni `socket`, `bind` e `listen`. Questo consente al server di ascoltare le connessioni in entrata dai client.
- **Inizializzazione degli Esami**: Una serie di esami predefiniti viene inizializzata, includendo ID, nomi e date degli esami.
- **Ciclo di Accettazione dei Client**: Il server rimane in ascolto per nuove connessioni. Per ogni client connesso, il server esegue una funzione specifica che gestisce le richieste del client in maniera sequenziale.
- **Chiusura del Server**: Al termine dell'operazione o in caso di interruzione, il socket del server viene chiuso per garantire una chiusura pulita delle risorse utilizzate.

Gestione Client con handle_client

La funzione `handle_client` gestisce la comunicazione con un singolo client. Quando un client si connette, il server utilizza questa funzione per interpretare e rispondere alle richieste del client.

- **Comunicazione con il Client**: Il server legge le richieste inviate dal client e, in base al contenuto, esegue le operazioni appropriate come la visualizzazione degli esami disponibili, la prenotazione di un esame, o l'aggiunta di una nuova data per un esame. Le risposte generate vengono poi inviate indietro al client.
- **Chiusura della Connessione**: Alla fine della comunicazione, il socket del client viene chiuso per terminare la connessione e liberare risorse.

Funzione Richiesta_Prenotazione

Questa funzione gestisce la prenotazione di un esame per un dato studente.

- **Verifica della Prenotazione Esistente**: Prima di procedere con la prenotazione, la funzione verifica se lo studente ha già prenotato lo stesso esame nella stessa data, prevenendo duplicazioni.
- **Aggiunta della Prenotazione**: Se non esiste una prenotazione per l'esame selezionato, la prenotazione viene aggiunta e la lista delle prenotazioni viene aggiornata.
- **Risposta al Client**: Una volta completata l'operazione, viene inviata una conferma al client con il numero totale di prenotazioni effettuate.

Funzione Esami_Disponibili

Questa funzione permette al client di visualizzare tutte le date disponibili per un determinato esame.

- **Filtraggio degli Esami**: La funzione filtra gli esami in base all'ID ricevuto dal client, mostrando solo le date corrispondenti.
- **Invio della Lista al Client**: La lista delle date disponibili viene formattata e inviata al client per la visualizzazione.

Funzione Aggiungi_Esame

Questa funzione gestisce l'inserimento di un nuovo esame o una nuova data per un esame esistente.

- **Verifica di Esami Esistenti**: Prima di aggiungere una nuova data, viene verificato se esiste già un esame con lo stesso ID e data per evitare duplicazioni.
- **Aggiunta della Nuova Data**: Se la data non è presente, la funzione aggiunge il nuovo esame alla lista e invia una conferma al client.

Scrittura Completa con FullWrite

La funzione `**FullWrite**` è utilizzata per garantire che tutti i dati siano inviati correttamente al client. Questa funzione gestisce l'invio su un socket, assicurandosi che l'intero buffer sia scritto, anche in presenza di interruzioni di sistema.

Comunicazione e Protocollo

La comunicazione tra i tre componenti avviene tramite socket TCP, con messaggi codificati in stringhe. Il protocollo di comunicazione è relativamente semplice, basato su comandi e risposte codificate:

- **Client Studente** invia comandi come "MostraEsami" o "PrenotaEsame".
- **Server Segreteria** inoltra questi comandi al **Server Universitario**, che li elabora e restituisce una risposta.
- Le risposte sono codificate in stringhe con informazioni specifiche come "EsamiDisponibili" o "ConfermaPrenotazione".

Diagramma di Flusso del Codice

Il codice segue una struttura gerarchica e sequenziale, con ogni componente (Client Studente, Server Segreteria, Server Universitario) che ha la propria responsabilità:

1**. Client Studente** → Richiede informazioni e prenotazioni.
2. **Server Segreteria** → Inoltra richieste e risposte tra il Client Studente e il Server Universitario.
3. **Server Universitario** → Gestisce i dati degli esami e le prenotazioni.

SEGRETERIA

SERVER_UNIVERSITARIO

1) Aggiungi esame

1) Scelta esame

Richiesta appelli esistenti

Appelli disponibili

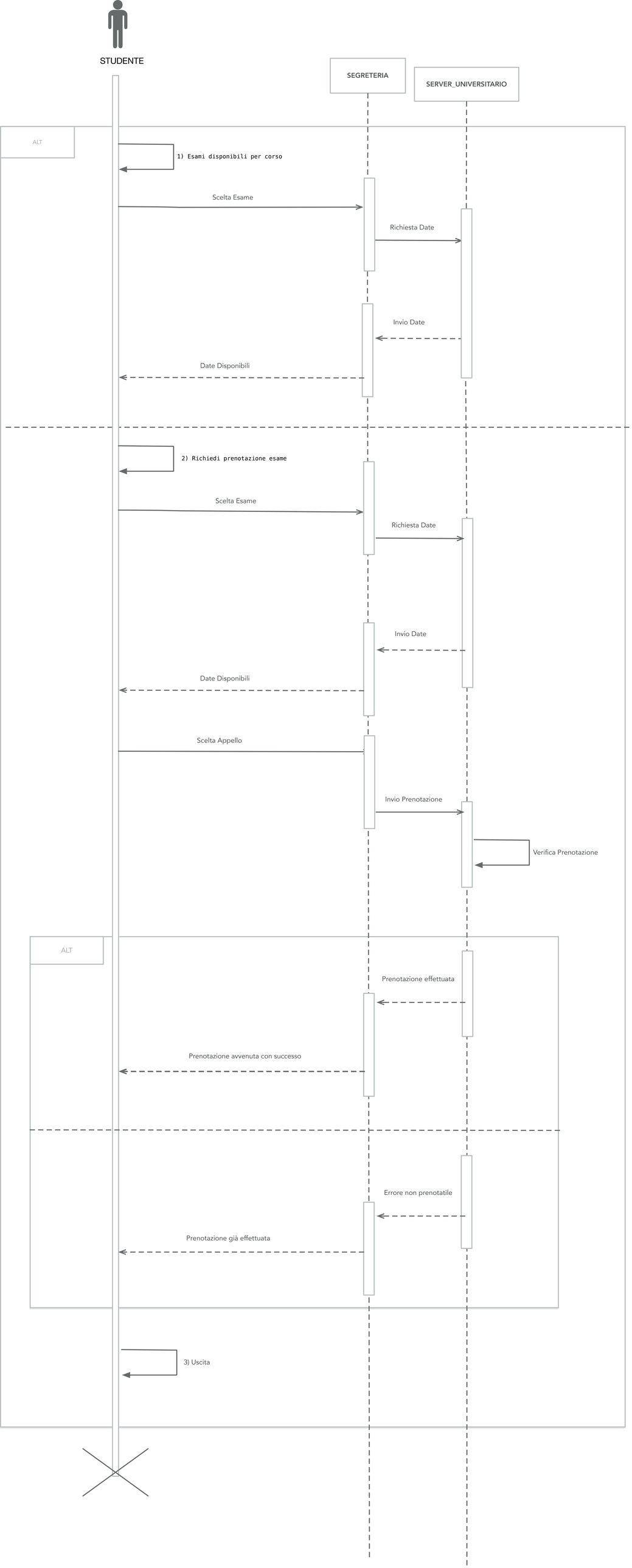
Inserimento data

Verifica inserimento data

ALT

Data corretta . Appello inserito

Data Errata , ritorno al menu


STUDENTE
SEGRETERIA
SERVER_UNIVERSITARIO
ALT
1) Esami disponibili per corso
Scelta Esame
Richiesta Date
Invio Date
Date Disponibili
2) Richiedi prenotazione esame
Scelta Esame
Richiesta Date
Invio Date
Date Disponibili
Scelta Appello
Invio Prenotazione
Verifica Prenotazione
ALT
Prenotazione effettuata
Prenotazione avvenuta con successo
Errore non prenotatile
Prenotazione già effettuata
3) Uscita

Gestione degli Errori
Il codice include controlli per gestire vari tipi di errori:
- Errori di connessione o di rete sono gestiti con messaggi di errore e la chiusura sicura del socket.
- Errori di input utente sono gestiti con messaggi di errore e richieste di reinserimento.

Guida all'Esecuzione
Compilazione
Prima di eseguire i programmi, devi compilarli. Di seguito sono riportati i comandi per compilare i file sorgente utilizzando il compilatore GCC:

Server Universitario:
gcc -o server_universitario Server_Universitario.c

Server Segreteria:
gcc -o server_segreteria Segreteria.c

Client Studente:
gcc -o client_studente Client.c

Esecuzione
Esecuzione del Server Universitario
Il server universitario è il componente principale che gestisce le richieste degli studenti e le prenotazioni degli esami. Avvia il server universitario con il seguente comando:

./server_universitario

Questo comando avvierà il server che ascolterà le richieste dei client sulla porta specificata (definita nella costante `PORT` nel codice).

Esecuzione del Server Segreteria

Il server segreteria potrebbe essere utilizzato per gestire una logica specifica o per implementare una funzionalità intermedia tra il client e il server universitario. Assicurati di avviare il server segreteria in una nuova finestra di terminale:

./server_segreteria

Esecuzione del Client Studente

Una volta che il server universitario è in esecuzione e ascolta le richieste, puoi avviare il client studente per interagire con il server. Usa il seguente comando per avviare il client:

./client_studente <indirizzo-ip-server>

Sostituisci `<indirizzo-ip-server>` con l'indirizzo IP del server universitario, che è di solito `127.0.0.1` per una connessione locale o l'indirizzo IP della macchina su cui il server è in esecuzione se usi una rete.

CODICE SORGENTE SEGRETERIA_STUDENTI

Inclusione delle librerie

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>
#include <ctype.h>
#include <stdbool.h>
```

- **sys/types.h**: Definisce i tipi di dati usati per la gestione di file e socket, come `size_t`, `ssize_t`, etc.
- **sys/socket.h**: Include definizioni e funzioni per la creazione e gestione dei socket.
- **netinet/in.h**: Contiene strutture e costanti per indirizzi Internet (IPv4), come `sockaddr_in`.
- **stdio.h**: Per input/output standard (es. `printf`, `fscanf`).
- **stdlib.h**: Include funzioni standard come `malloc`, `exit`.
- **string.h**: Gestisce operazioni su stringhe (es. `memcpy`, `strlen`).
- **unistd.h**: Contiene funzioni come `read`, `write`, e altre utilità UNIX (es. `close` per chiudere i file descriptor).
- **pthread.h**: Per la gestione dei thread POSIX e dei mutex.
- **errno.h**: Gestisce errori a livello di sistema, associando codici di errore globali (es. `errno`).
- **ctype.h**: Per funzioni di controllo dei caratteri (`isalpha`, `isdigit`, etc.).
- **stdbool.h**: Fornisce il tipo booleano `bool` con i valori `true` e `false`.

Definizione di costanti

```
#define PORT 54321 // Porta su cui il server ascolterà le connessioni
#define MAXLINE 1024 // Dimensione massima dei buffer
```

- **PORT**: Il server ascolterà sulla porta `54321`. Questa porta sarà utilizzata per stabilire connessioni tra client e server.
- **MAXLINE**: La dimensione massima del buffer è definita come 1024 byte. Questo buffer sarà usato per trasferire dati.

Dichiarazione delle variabili globali

```
int university_socket = -1;
int server_socket = -1;
int keep_running = 1;
```

- **university_socket**: Socket usato per comunicare con un server esterno, presumibilmente dell'università. È inizializzato a `-1` per indicare che non è ancora stato creato o connesso.
- **server_socket**: Socket principale che il server utilizzerà per accettare connessioni dai client. Anch'esso è inizialmente impostato a `-1`.
- **keep_running**: Variabile di controllo per gestire l'esecuzione continua del server. Il server continuerà a funzionare finché questa variabile è impostata a `1`.

Mutex per la sincronizzazione

```
pthread_mutex_t university_socket_mutex = PTHREAD_MUTEX_INITIALIZER;
```

- **university_socket_mutex**: È un mutex (mutual exclusion) usato per gestire l'accesso concorrente al socket universitario. Un mutex garantisce che solo un thread alla volta possa modificare o accedere alla risorsa protetta, in questo caso il socket universitario.

Funzione `FullWrite`

```
ssize_t FullWrite(int fd, const void *buf, size_t count) {
    size_t nleft = count;
    ssize_t nwritten;
    const char *ptr = buf;
```

- **FullWrite**: Questa funzione è progettata per inviare tutti i dati contenuti in un buffer a un file descriptor (come un socket o un file).
  - **fd**: Il file descriptor a cui inviare i dati.
  - **buf**: Puntatore al buffer contenente i dati da inviare.
  - **count**: Numero di byte da inviare.

Funzione `FullWrite` - Scrive completamente un buffer su un file descriptor

```
while (nleft > 0) {
    if ((nwritten = write(fd, ptr, nleft)) <= 0) {
        if (nwritten < 0 && errno == EINTR) {
```

```
            continue; // Riprovare se l'operazione è interrotta da un segnale
        } else {
            return -1; // Errore non recuperabile, interrompe la scrittura
        }
    }
    nleft -= nwritten; // Aggiorna il numero di byte rimanenti da scrivere
    ptr += nwritten;   // Sposta il puntatore nel buffer per indicare dove
riprendere la scrittura
}
return count;
```

- **nleft**: Viene inizializzato con il numero di byte totali da scrivere (`count`). Serve a tenere traccia di quanti byte rimangono ancora da scrivere.
- **ptr**: Puntatore al buffer `buf` . Viene usato per avanzare nel buffer man mano che i dati vengono scritti.
  `while (nleft > 0)` :
  - Questo ciclo `while` continua finché ci sono ancora byte da scrivere nel buffer. La variabile `nleft` tiene traccia del numero di byte rimanenti da inviare.
  - Se `nleft` è maggiore di zero, il ciclo tenta di scrivere i byte rimanenti.

`write(fd, ptr, nleft)` :
- Questa funzione di sistema scrive fino a `nleft` byte dal buffer (puntato da `ptr`) nel file descriptor `fd` (che potrebbe rappresentare una connessione di rete o un file).
- `nwritten` : Numero di byte che sono stati scritti effettivamente. Può restituire un numero inferiore a `nleft` se, ad esempio, il socket è temporaneamente non in grado di accettare tutti i dati (questo è comune nelle operazioni non bloccanti).
- Se `nwritten <= 0` , può significare che si è verificato un errore o che la scrittura è stata interrotta.

Gestione degli errori:
- **Errore `EINTR` (Interruzione da segnale)**: Se `write` viene interrotto da un segnale, la chiamata può fallire con l'errore `EINTR` . In questo caso, si continua a riprovare senza interrompere il ciclo (`continue`).
- **Altri errori**: Se l'errore non è `EINTR` , il programma restituisce `-1` , segnalando che la scrittura è fallita.

Aggiornamento delle variabili:
- `nleft -= nwritten` : Dopo una scrittura parziale, viene sottratto il numero di byte effettivamente scritti da `nleft` . Questo riduce il conteggio dei byte che rimangono da scrivere.
- `ptr += nwritten` : Il puntatore `ptr` viene avanzato di `nwritten` byte per indicare la nuova posizione nel buffer da cui riprendere la scrittura.

Restituzione finale:
- Una volta che tutto il buffer è stato scritto (`nleft == 0`), la funzione ritorna il valore iniziale di `count` , che è il numero totale di byte che la funzione doveva scrivere.

Funzione `is_digit_string` - Verifica se una stringa contiene solo numeri

```
bool is_digit_string(const char *str, size_t length) {
    for (size_t i = 0; i < length; i++) {
        if (!isdigit(str[i])) {
            return false;
        }
    }
    return true;
}
```

`for (size_t i = 0; i < length; i++)` :
- Un ciclo `for` che attraversa ogni carattere della stringa `str` fino a raggiungere `length` (la lunghezza della stringa da controllare).

`isdigit(str[i])` :
- Per ogni carattere `str[i]` , la funzione `isdigit` viene utilizzata per verificare se il carattere è una cifra numerica (0-9).
- Se la funzione `isdigit` restituisce `false` per un qualsiasi carattere (quindi il carattere non è una cifra), il ciclo esce e la funzione ritorna `false` .

`return false;` :
- Se viene trovato un carattere non numerico, la funzione termina immediatamente e restituisce `false` , indicando che la stringa non contiene solo cifre.

`return true;` :
- Se tutti i caratteri della stringa sono cifre numeriche, il ciclo termina correttamente e la funzione ritorna `true` , indicando che l'intera stringa è composta da numeri.

Funzione `is_valid_day`

La funzione `is_valid_day` verifica se una data specificata (giorno, mese, anno) è valida, tenendo conto del numero di giorni per ogni mese e se l'anno è bisestile.

```
bool is_valid_day(int day, int month, int year) {
    if (month < 1 || month > 12) {
        return false;
    }
```

```
    if (day < 1 || day > 31) {
        return false;
    }
```

Verifica del mese (`month`):

- Si controlla che il valore di `month` sia compreso tra 1 e 12. Se il valore non è valido, la funzione ritorna `false`, indicando che la data è invalida.

Verifica del giorno (`day`):

- Similmente, si verifica che il valore di `day` sia compreso tra 1 e 31. Se il valore non è valido, la funzione ritorna immediatamente `false`.

```
    // Mesi con 30 giorni
    if (month == 4 || month == 6 || month == 9 || month == 11) {
        if (day > 30) {
            return false;
        }
    }
```

Mesi con 30 giorni:

- I mesi di aprile (4), giugno (6), settembre (9) e novembre (11) hanno solo 30 giorni.
- Se il mese è uno di questi e il giorno è maggiore di 30, la funzione ritorna `false`, poiché una data come "31 aprile" non è valida.

```
    // Febbraio
    if (month == 2) {
        bool is_leap = (year % 4 == 0 && (year % 100 != 0 || year % 400 == 0));
        if ((is_leap && day > 29) || (!is_leap && day > 28)) {
            return false;
        }
    }
```

Gestione del mese di febbraio:

- Per il mese di febbraio, si tiene conto dell'anno bisestile. Un anno è bisestile se è divisibile per 4, ma non per 100, a meno che non sia anche divisibile per 400 (regola degli anni bisestili).
- Se il mese è febbraio e l'anno è bisestile, il giorno non può essere maggiore di 29. Se l'anno non è bisestile, il giorno non può superare 28.
- Se queste condizioni non sono rispettate, la funzione ritorna `false`.

```
    return true;
}
```

Restituzione finale:

- Se tutte le verifiche sono superate (mese valido, giorno valido, controlli su febbraio e sui mesi con 30 giorni), la funzione ritorna `true`, indicando che la data è valida.

Funzione `normalize_date_parts`

La funzione `normalize_date_parts` prende le parti di una data (giorno e mese come stringhe) e le "normalizza", ovvero le converte in un formato a due cifre, ad esempio "1" diventa "01".

```
void normalize_date_parts(char *day_str, char *month_str) {
    int day = atoi(day_str);
    int month = atoi(month_str);
```

Conversione da stringa a intero:

- La funzione inizia convertendo le stringhe `day_str` e `month_str` in numeri interi usando la funzione `atoi` (ASCII to Integer). Questo è necessario per poter controllare i valori numerici e formattarli correttamente.

```
    // Normalizzazione del giorno
    if (day >= 1 && day <= 9) {
        snprintf(day_str, 3, "0%d", day);
    } else {
        snprintf(day_str, 3, "%d", day);
    }
```

Normalizzazione del giorno:

- Se il giorno è compreso tra 1 e 9, viene aggiunto uno zero all'inizio, utilizzando `snprintf`. Il nuovo valore viene scritto in `day_str`.
- Se il giorno è già compreso tra 10 e 31, viene mantenuto così com'è.

```
    // Normalizzazione del mese
    if (month >= 1 && month <= 9) {
        snprintf(month_str, 3, "0%d", month);
    } else {
```

```
        snprintf(month_str, 3, "%d", month);
    }
}
```

Normalizzazione del mese:

- La stessa logica viene applicata al mese. Se il mese è compreso tra 1 e 9, si aggiunge uno zero all'inizio. Altrimenti, il mese viene mantenuto così com'è.

Funzione `validate_date`

La funzione `validate_date` prende tre stringhe come input: `day_str`, `month_str` e `year_str`. Queste rappresentano il giorno, il mese e l'anno, rispettivamente. La funzione esegue una serie di controlli e confronti per determinare se la data è valida e nel formato corretto.

Copia e gestione delle stringhe del giorno e del mese

```
char day_copy[3];
char month_copy[3];
strncpy(day_copy, day_str, 2);
strncpy(month_copy, month_str, 2);
day_copy[2] = '\0';
month_copy[2] = '\0';
```

- **Copia delle stringhe**: Si creano due buffer di caratteri, `day_copy` e `month_copy`, ciascuno lungo 3 caratteri. Anche se il giorno e il mese sono espressi in massimo 2 caratteri, vengono aggiunti un terzo carattere `\0` per terminare la stringa in C (in C le stringhe devono sempre essere terminate da `\0` per essere riconosciute come stringhe valide).
- **Uso di `strncpy`**: La funzione `strncpy` copia i primi 2 caratteri delle stringhe `day_str` e `month_str` nei buffer `day_copy` e `month_copy`. `strncpy` è utilizzata per assicurare che non si copi accidentalmente più di 2 caratteri (in caso le stringhe originali siano più lunghe).
- **Aggiunta del terminatore**: Dopo la copia, il carattere di terminazione `\0` è aggiunto esplicitamente al terzo posto di entrambe le stringhe (`day_copy[2]` e `month_copy[2]`), assicurandosi che siano stringhe valide in C.

Normalizzazione delle parti della data

```
normalize_date_parts(day_copy, month_copy);
```

La funzione `normalize_date_parts` ha il compito di "normalizzare" le stringhe del giorno e del mese. La normalizzazione potrebbe comportare operazioni come:

- **Aggiunta di zeri iniziali**: Ad esempio, se il giorno fosse rappresentato come `"5"`, potrebbe essere convertito in `"05"`.
- **Rimozione di spazi o caratteri non validi**: La funzione potrebbe rimuovere eventuali spazi vuoti o altri caratteri non numerici.****

Verifica della validità delle stringhe numeriche

```
if (!is_digit_string(day_copy, 2) || !is_digit_string(month_copy, 2) ||
!is_digit_string(year_str, 4)) {
    return false;
}
```

Dopo la normalizzazione, si verifica che le stringhe `day_copy`, `month_copy` e `year_str` siano effettivamente formate da cifre e abbiano la lunghezza corretta.

- **Funzione `is_digit_string`**: Questa funzione controlla che:
  - La stringa sia composta esclusivamente da caratteri numerici.
  - La stringa abbia la lunghezza specificata (2 per `day_copy` e `month_copy`, 4 per `year_str`).

Se una delle stringhe non soddisfa queste condizioni, la funzione restituisce immediatamente `false`, segnalando che la data non è valida.

Conversione delle stringhe in numeri

```
int day = atoi(day_copy);
int month = atoi(month_copy);
int year = atoi(year_str);
```

- `atoi`: Converte le stringhe `day_copy`, `month_copy` e `year_str` in numeri interi.
  - `day`: Il valore del giorno convertito da `day_copy`.
  - `month`: Il valore del mese convertito da `month_copy`.
  - `year`: Il valore dell'anno convertito da `year_str`.

Ottenere la data corrente

```
time_t t = time(NULL);
struct tm *tm_info = localtime(&t);
```

Qui si ottiene la data corrente utilizzando funzioni di libreria standard:

- `time(NULL)`: Restituisce il numero di secondi trascorsi dal 1° gennaio 1970 (l'epoca Unix). Viene usato per ottenere il tempo corrente.
- `localtime(&t)`: Questa funzione converte il numero di secondi in una struttura `tm`, che contiene informazioni dettagliate sulla data e l'ora locali. Ad esempio, contiene l'anno, il mese, il giorno, l'ora, e così via.

Calcolo dell'anno, mese e giorno correnti

```
int current_year = tm_info->tm_year + 1900;
int current_month = tm_info->tm_mon + 1;
int current_day = tm_info->tm_mday;
```

- Dopo aver ottenuto la data corrente, si estraggono l'anno, il mese e il giorno dalla struttura `tm`:
- `tm_year`: Rappresenta il numero di anni trascorsi dal 1900, quindi si aggiunge 1900 per ottenere l'anno corrente.
  - Esempio: Se `tm_year` fosse `123`, l'anno corrente sarebbe `1900 + 123 = 2023`.
- `tm_mon`: Rappresenta i mesi dell'anno come un numero da 0 (gennaio) a 11 (dicembre). Si aggiunge 1 per ottenere il mese nel formato da 1 a 12.
  - Esempio: Se `tm_mon` fosse `9`, il mese corrente sarebbe `9 + 1 = 10` (ottobre).
- `tm_mday`: Contiene il giorno del mese corrente (da 1 a 31).

Confronto tra la data fornita e la data corrente

```
if (year < current_year) {
    return false;
} else if (year == current_year) {
    if (month < current_month) {
        return false;
    } else if (month == current_month) {
        if (day <= current_day) {
            return false;
        }
    }
}
```

Questo è il punto cruciale del controllo. La funzione confronta la data fornita con la data corrente, per verificare che la data inserita sia **nel futuro** rispetto alla data corrente.

- **Controllo dell'anno**:
  - Se l'anno della data fornita (`year`) è inferiore all'anno corrente (`current_year`), la data è passata, quindi la funzione restituisce `false` .
- **Controllo del mese**:
  - Se l'anno fornito è lo stesso di quello corrente, si confronta il mese. Se il mese della data fornita (`month`) è inferiore al mese corrente (`current_month`), la data è passata e si restituisce `false` .
- **Controllo del giorno**:
  - Se l'anno e il mese coincidono con quelli attuali, si confronta il giorno. Se il giorno della data fornita (`day`) è inferiore o uguale al giorno corrente (`current_day`), la data è passata e si restituisce `false` .

Verifica della validità del giorno

```
return is_valid_day(day, month, year);
```

Se la data fornita non è stata scartata dai controlli precedenti (cioè non è una data passata), l'ultimo passo è verificare che il giorno sia valido per il mese e l'anno specificati:

- **Funzione** `is_valid_day` : Questa funzione verifica se il giorno fornito è un giorno valido per quel mese e quell'anno. Ad esempio:
  - Il giorno non deve superare il numero massimo di giorni del mese (ad esempio, 30 giorni per novembre, 31 giorni per dicembre).
  - Se il mese è febbraio, la funzione deve controllare se l'anno è bisestile, poiché febbraio può avere 29 giorni in un anno bisestile.

Approfondimento strncpy

La funzione `strncpy` è una versione "limitata" di `strcpy` , che copia una stringa da una sorgente a una destinazione, ma con un limite massimo di caratteri da copiare. La sua firma è la seguente:

```
char *strncpy(char *dest, const char *src, size_t n);
```

- `dest` : Il puntatore alla destinazione dove verrà copiata la stringa.
- `src` : Il puntatore alla stringa sorgente da cui si copieranno i caratteri.

- `n`: Il numero massimo di caratteri da copiare da `src` a `dest`.

Comportamento di `strncpy`

Copia dei caratteri: `strncpy` copia fino a `n` caratteri dalla stringa `src` nella stringa `dest`.
Terminazione con `\0`:

- Se la lunghezza di `src` è inferiore a `n`, `strncpy` aggiunge automaticamente caratteri nulli (`\0`) alla fine di `dest` per completare la dimensione di `n`.
- **Se `src` è lunga almeno `n`, `strncpy` non aggiunge il carattere di terminazione (`\0`) a `dest`**. In questo caso, devi aggiungerlo manualmente alla fine della stringa copiata se necessario.
  Ritaglio della stringa: Se `src` è più lunga di `n`, vengono copiati solo i primi `n` caratteri, e il resto della stringa sorgente viene ignorato.

Dettagli importanti:

- **Non sempre termina con `\0`**: Come detto, se `n` è uguale o maggiore alla lunghezza di `src`, `strncpy` riempie il resto di `dest` con `\0`. Se `n` è minore della lunghezza di `src`, `dest` **non sarà terminata con `\0`**, quindi è buona norma aggiungere manualmente il carattere nullo.
- **Riempimento con `\0`**: Se la lunghezza di `src` è inferiore a `n`, tutti i caratteri dopo la copia vengono riempiti con caratteri nulli per garantire che `dest` abbia esattamente `n` caratteri.

Uso tipico di `strncpy`

- Viene utilizzato quando si vuole copiare una stringa limitando la quantità di caratteri, ad esempio per prevenire il sovraccarico di buffer o per lavorare con buffer di dimensione fissa.

Elabora e formatta una data nel formato "dd-mm-yyyy"

Il codice implementa una funzione chiamata `process_date` che prende in input tre stringhe rappresentanti il giorno, il mese e l'anno, verifica se la data è valida e, se lo è, la formatta nel formato `"dd-mm-yyyy"`. Di seguito è riportata una descrizione approfondita di ogni parte del codice.

```
// Elabora e formatta una data nel formato "dd-mm-yyyy"

char* process_date(const char *giorno, const char *mese, const char *anno, int
*should_return) {
```

```
    static char formatted_date[11]; // Spazio sufficiente per "dd-mm-yyyy\0"
    char formatted_day[3], formatted_month[3];

    // Normalizza giorno e mese con zeri iniziali
    snprintf(formatted_day, sizeof(formatted_day), "%02d", atoi(giorno));
    snprintf(formatted_month, sizeof(formatted_month), "%02d", atoi(mese));

    // Verifica se la data è valida
    if (!validate_date(formatted_day, formatted_month, anno)) {
        printf("Data non valida.\n");
        *should_return = 1;
        return NULL;
    }

    // Unisci giorno, mese e anno in una singola stringa
    snprintf(formatted_date, sizeof(formatted_date), "%s-%s-%s", formatted_day,
formatted_month, anno);
    *should_return = 0; // Data valida
    return formatted_date;
}
```

Dichiarazione e preparazione della stringa formattata

```
static char formatted_date[11]; // Spazio sufficiente per "dd-mm-yyyy\0"
char formatted_day[3], formatted_month[3];
```

- `static char formatted_date[11]`:
  - **Spazio per la data formattata**: La variabile `formatted_date` è dichiarata come `static`, quindi mantiene il suo valore tra più chiamate alla funzione. È lunga 11 caratteri, abbastanza per contenere una data nel formato `dd-mm-yyyy` (10 caratteri) e il terminatore nullo `\0`.
  - **Memoria persistente**: Poiché è `static`, il buffer esiste finché il programma è in esecuzione e non viene cancellato al termine della funzione.
- `char formatted_day[3], formatted_month[3]`:
  - Questi array di 3 caratteri sono usati per contenere la versione formattata a due cifre del giorno e del mese (es. `01`, `09`, etc.), più il carattere nullo alla fine.

Normalizzazione del giorno e del mese

```
snprintf(formatted_day, sizeof(formatted_day), "%02d", atoi(giorno));
snprintf(formatted_month, sizeof(formatted_month), "%02d", atoi(mese));
```

- `snprintf(formatted_day, sizeof(formatted_day), "%02d", atoi(giorno))`:
  - `atoi(giorno)`: Converte la stringa `giorno` in un numero intero.
  - `"%02d"`: Il formato `"%02d"` forza il numero a essere rappresentato con almeno due cifre. Se il numero è inferiore a 10, viene aggiunto uno zero davanti (ad esempio, `1` diventa `01`).
  - `snprintf`: Scrive la stringa formattata (due cifre) nell'array `formatted_day` con una lunghezza massima di 2 caratteri, più il terminatore nullo (`\0`). Lo stesso procedimento viene eseguito per il mese (`mese`).

Validazione della data

```
if (!validate_date(formatted_day, formatted_month, anno)) {
    printf("Data non valida.\n");
    *should_return = 1;
    return NULL;
}
```

- `validate_date(formatted_day, formatted_month, anno)`:
  - Chiama la funzione `validate_date` per verificare se la data composta da `formatted_day`, `formatted_month` e `anno` è valida. Questa funzione controlla la validità del giorno, del mese e dell'anno, tenendo conto degli anni bisestili e dei giorni validi per ogni mese.
- **Se la data non è valida**:
  - Viene stampato il messaggio `"Data non valida."` per informare l'utente.

- `*should_return = 1`: Viene impostato il valore di `should_return` a `1` per segnalare che c'è stato un errore e la data non è valida.
- La funzione termina restituendo `NULL`, poiché non è possibile restituire una data valida.

Formattazione finale della data

```
snprintf(formatted_date, sizeof(formatted_date), "%s-%s-%s", formatted_day,
formatted_month, anno);
*should_return = 0; // Data valida
return formatted_date;
```

- **Formattazione della data completa**:
  - Se la data è valida, viene composta una stringa finale usando `snprintf`.
  - `"%s-%s-%s"`: Il giorno, il mese e l'anno (ora tutti in formato stringa) vengono concatenati usando il formato `"%s-%s-%s"`, che produce una stringa nel formato `dd-mm-yyyy`.
  - La stringa viene scritta nel buffer `formatted_date`.
- **Segnalazione di successo**:
  - `*should_return = 0`: Imposta `should_return` a `0` per indicare che la data è valida.
- **Restituzione della data**:
  - La funzione restituisce `formatted_date`, che contiene la data formattata.

Gestisce l'inserimento e l'invio della data per un esame

Il codice implementa la funzione `handle_exam_date`, la quale gestisce l'inserimento e l'invio della data di un esame da parte dell'utente. La funzione interagisce con l'utente per ottenere una data e poi valida e invia la data se è corretta. Di seguito è riportata una descrizione approfondita di ogni parte del codice.

```
void handle_exam_date(const char *exam_name, const char *exam_code, int exam_type,
int *should_return, int university_socket) {

    char giorno[3], mese[3], anno[5];

    char buffer[MAXLINE];

    int nwrite;



    system("clear");
```

```
printf("Inserimento data di esame di %s: \n", exam_name);


strcpy(buffer, exam_code);
strcat(buffer, "_1");
nwrite = FullWrite(university_socket, buffer, strlen(buffer));
if (nwrite < 0) {
        printf("Errore in scrittura\n");
        return;
    }


// Legge e stampa la lista degli esami disponibili
char recvbuff[MAXLINE];
ssize_t nread = read(university_socket, recvbuff, sizeof(recvbuff) - 1);
if (nread < 0) {
    perror("Errore in lettura");
    return;
}
recvbuff[nread] = '\0'; // Termina la stringa
fputs(recvbuff, stdout); // Stampa la risposta del server
printf("\n");


printf("Inserisci il giorno: ");
scanf("%2s", giorno);
```

```
printf("Inserisci il mese: ");
scanf("%2s", mese);

printf("Inserisci l'anno: ");
scanf("%4s", anno);

if (!validate_date(giorno, mese, anno)) {
    system("clear");
    printf("Data non valida.\n");
    *should_return = 1;
    return;
}

// Unisci giorno, mese e anno in una singola stringa
char *date = process_date(giorno, mese, anno, should_return);

system("clear");
printf("SEGRETERIA) Esame: %s Data: %s Aggiunto\n", exam_name, date);

snprintf(buffer, sizeof(buffer), "%s:%s_%d", exam_code, date, exam_type);
nwrite = FullWrite(university_socket, buffer, strlen(buffer));
```

```
    if (nwrite < 0) {
        printf("Errore in scrittura: %s\n", strerror(errno));
        return;
    }

    nread = read(university_socket, recvbuff, sizeof(recvbuff) - 1);
    if (nread < 0) {
        perror("Errore in lettura");
        return;
    }
    recvbuff[nread] = '\0';
    fputs(recvbuff, stdout);
    printf("\n");

    *should_return = 1; // Imposta il flag per tornare al menu principale
}
```

Dichiarazione delle variabili locali

```
char giorno[3], mese[3], anno[5];
char buffer[MAXLINE];
int nwrite;
```

- **`giorno[3], mese[3], anno[5]`**: Array di caratteri per memorizzare le parti della data:
  - `giorno`: Per il giorno (es. "09"), massimo 2 caratteri più il terminatore `\0`.
  - `mese`: Per il mese (es. "04"), massimo 2 caratteri più il terminatore `\0`.
  - `anno`: Per l'anno (es. "2024"), massimo 4 caratteri più il terminatore `\0`.

- `buffer[MAXLINE]`: Buffer usato per contenere i dati da inviare tramite il socket. La dimensione è determinata da `MAXLINE`, che viene definito altrove (probabilmente come 1024 byte).
- `nwrite`: Variabile per tenere traccia del numero di byte scritti nel socket.

Pulizia dello schermo e richiesta dell'input

```
system("clear");
printf("Inserimento data di esame di %s: \n", exam_name);
```

- `system("clear")`: Usa il comando `clear` per pulire lo schermo della console, migliorando la leggibilità per l'utente.
- `printf("Inserimento data di esame di %s: \n", exam_name)`: Stampa un messaggio che indica all'utente che sta inserendo la data per un esame specifico. Il nome dell'esame viene passato come argomento alla funzione (`exam_name`).

Raccolta dell'input dal terminale

```
printf("Inserisci il giorno: ");
scanf("%2s", giorno);

printf("Inserisci il mese: ");
scanf("%2s", mese);

printf("Inserisci l'anno: ");
scanf("%4s", anno);
```

- `scanf("%2s", giorno)`: Legge una stringa di massimo 2 caratteri inserita dall'utente e la salva nella variabile `giorno`. Viene limitata a 2 caratteri per evitare di eccedere la dimensione dell'array.
- `scanf("%2s", mese)`: Stessa cosa per il mese, salva l'input nella variabile `mese`.
- `scanf("%4s", anno)`: Legge fino a 4 caratteri per l'anno e li salva in `anno`.

> Nota: Non viene gestito l'overflow dell'input dall'utente. Se l'utente inserisce più di 2 o 4 caratteri, potrebbero verificarsi problemi, quindi potrebbe essere utile gestire questo aspetto in modo più robusto.

Validazione della data

```
if (!validate_date(giorno, mese, anno)) {
    system("clear");
```

```
    printf("Data non valida.\n");
    *should_return = 1;
    return;
}
```

- `validate_date(giorno, mese, anno)`: Questa funzione verifica se la combinazione giorno, mese e anno è una data valida. Usa il giorno, mese e anno forniti dall'utente e controlla se sono coerenti (tenendo conto degli anni bisestili, dei giorni disponibili nei vari mesi, ecc.).
- **Se la data non è valida**:
  - `system("clear")`: Cancella nuovamente lo schermo per rimuovere l'input precedente.
  - `printf("Data non valida.\n")`: Stampa un messaggio di errore informando l'utente che la data inserita non è corretta.
  - `*should_return = 1`: Imposta il puntatore `should_return` a 1 per segnalare che c'è stato un errore. Il valore passato viene modificato per segnalare che l'operazione non è andata a buon fine.
  - `return`: Termina la funzione, interrompendo ulteriori operazioni.

Unione di giorno, mese e anno in una singola stringa

```
char *date = process_date(giorno, mese, anno, should_return);
```

- `process_date(giorno, mese, anno, should_return)`: Chiama la funzione `process_date`, che:
  - Converte `giorno`, `mese`, e `anno` in stringhe nel formato corretto (ad esempio, "09-04-2024").
  - Valida la data e restituisce una stringa formattata nel formato `"dd-mm-yyyy"`.
  - Se la data non è valida, imposta `*should_return` a `1` per segnalare che l'operazione deve terminare.

Se la data è valida, `date` conterrà la stringa `"dd-mm-yyyy"`.

Pulizia dello schermo e stampa della conferma

```
system("clear");
printf("SEGRETERIA) Esame: %s Data: %s Aggiunto\n", exam_name, date);
```

- `system("clear")`: Pulisce lo schermo della console, migliorando la leggibilità per l'utente.
- `printf(...)`: Stampa un messaggio di conferma per l'utente, mostrando il nome dell'esame (`exam_name`) e la data dell'esame (`date`), ad esempio:

- `"SEGRETERIA) Esame: Matematica Data: 09-04-2024 Aggiunto"`

Formattazione dei dati da inviare al server

```
snprintf(buffer, sizeof(buffer), "%s:%s_%d", exam_code, date, exam_type);
```

- **`snprintf(buffer, sizeof(buffer), "%s:%s_%d", exam_code, date, exam_type)`** :
  - Crea una stringa nel formato `"exam_code:date_exam_type"` e la memorizza in `buffer`. Esempio:
    - **`exam_code`**: Il codice dell'esame, ad esempio "MATH101".
    - **`date`**: La data formattata dall'utente, ad esempio "09-04-2024".
    - **`exam_type`**: Tipo di esame, rappresentato da un intero (`exam_type`).
    - Esempio finale: `"MATH101:09-04-2024_1"`
- La stringa creata sarà inviata al server per registrare l'esame, utilizzando il socket.

Invio dei dati tramite il socket

```
nwrite = FullWrite(university_socket, buffer, strlen(buffer));
```

- **`FullWrite(university_socket, buffer, strlen(buffer))`** :
  - Scrive i dati contenuti in `buffer` sul socket `university_socket`, assicurandosi di inviare tutti i byte.
  - **`FullWrite`**: È una funzione che garantisce che tutti i dati vengano inviati completamente, gestendo eventuali interruzioni (come errori temporanei o scritture parziali).
  - **`nwrite`**: Il numero di byte effettivamente inviati al server.

Gestione degli errori di scrittura

```
if (nwrite < 0) {
    printf("Errore in scrittura: %s\n", strerror(errno));
    return;
}
```

- **Controllo di errori**: Se si verifica un errore durante l'invio dei dati (se `nwrite` è negativo), viene stampato un messaggio di errore contenente il motivo (usando `strerror(errno)`), che restituisce una stringa descrittiva dell'errore.
- **Uscita anticipata**: In caso di errore, la funzione termina senza ulteriori azioni.

Lettura della risposta dal server

```
char recvbuff[MAXLINE];
ssize_t nread = read(university_socket, recvbuff, sizeof(recvbuff) - 1);
if (nread < 0) {
    perror("Errore in lettura");
    return;
}
recvbuff[nread] = '\0';
```

- `read(university_socket, recvbuff, sizeof(recvbuff) - 1)`:
  - Legge la risposta dal server attraverso il socket `university_socket` e la memorizza in `recvbuff`.
  - `sizeof(recvbuff) - 1`: Assicura che lo spazio per il terminatore nullo `\0` sia disponibile.
  - `nread`: Numero di byte letti dal socket.
- **Controllo di errori in lettura**:
  - Se si verifica un errore durante la lettura (`nread < 0`), viene stampato un messaggio d'errore usando `perror` e la funzione termina.
- **Aggiunta del terminatore nullo**:
  - Dopo aver letto i dati, si imposta manualmente il terminatore nullo alla fine della stringa ricevuta (`recvbuff[nread] = '\0'`), assicurando che `recvbuff` sia una stringa valida.

Visualizzazione della risposta del server

```
fputs(recvbuff, stdout);
printf("\n");
```

- `fputs(recvbuff, stdout)`:
  - Stampa la risposta del server (memorizzata in `recvbuff`) direttamente sulla console. `fputs` viene utilizzato per scrivere la stringa ricevuta sullo `stdout`.
  - Questo permette all'utente di vedere la risposta, che potrebbe essere una conferma dell'avvenuta registrazione dell'esame o un messaggio di errore dal server.

Aggiornamento del flag per tornare al menu principale

```
*should_return = 1; // Imposta il flag per tornare al menu principale
```

- `*should_return = 1`: Dopo che l'operazione è stata completata (o fallita), viene impostato il valore di `should_return` a `1` per segnalare che la funzione ha terminato e il programma può tornare al menu principale o a un'altra operazione successiva.

Funzione per l'inserimento dell'esame tramite l'interfaccia utente

Il codice fornisce un'interfaccia utente per selezionare un esame, inserire la data e inviare i dettagli al server tramite un socket. Di seguito viene descritta ogni parte del codice in modo approfondito.

```
void Inserisci_esame(int university_socket, int *should_return) {

    int scelta;



    while (1) {

        system("clear");

        printf("Date Esami:\n");

        printf("Scegliere l'esame:\n");

        printf("1) Reti di Calcolatori\n");

        printf("2) Algoritmi e Strutture Dati\n");

        printf("3) Programmazione 1\n");

        printf("4) Programmazione 2\n");

        printf("5) Programmazione 3\n");

        printf("6) Tecnologie Web\n");

        printf("0) Torna Indietro\n");



        if (scanf("%d", &scelta) != 1) {

            printf("Input non valido.\n");

            while (getchar() != '\n'); // Pulisce il buffer di input
```

```
            continue;
        }


        switch (scelta) {
            case 1:
                handle_exam_date("Reti di Calcolatori", "Reti", 3, should_return,
university_socket);
                break;
            case 2:
                handle_exam_date("Algoritmi e Strutture Dati", "Algoritmi", 3,
should_return, university_socket);
                break;
            case 3:
                handle_exam_date("Programmazione 1", "Prog1", 3, should_return,
university_socket);
                break;
            case 4:
                handle_exam_date("Programmazione 2", "Prog2", 3, should_return,
university_socket);
                break;
            case 5:
                handle_exam_date("Programmazione 3", "Prog3", 3, should_return,
university_socket);
                break;
            case 6:
```

```
                handle_exam_date("Tecnologie Web", "Web", 3, should_return,
university_socket);
                break;
            case 0:
                *should_return = 1; // Torna al menu principale
                return;
            default:
                system("clear");
                printf("Opzione non disponibile\n");
                *should_return = 1;
                return;
        }

        if (should_return) {
            return; // Esci dalla funzione se necessario
        }
    }
}
```

Dichiarazione delle variabili

```
int scelta;
```

- `scelta`: Questa variabile memorizza l'opzione scelta dall'utente (un numero intero), che sarà utilizzata per determinare quale esame l'utente desidera inserire.

Loop principale

```c
while (1) {
    system("clear");
    printf("Date Esami:\n");
    printf("Scegliere l'esame:\n");
    printf("1) Reti di Calcolatori\n");
    printf("2) Algoritmi e Strutture Dati\n");
    printf("3) Programmazione 1\n");
    printf("4) Programmazione 2\n");
    printf("5) Programmazione 3\n");
    printf("6) Tecnologie Web\n");
    printf("0) Torna Indietro\n");
```

- `while (1)`: Il loop infinito permette all'utente di scegliere ripetutamente tra le opzioni finché non decide di tornare indietro (scegliendo l'opzione `0`).
- `system("clear")`: Cancella lo schermo della console per dare una visualizzazione pulita prima di ogni ciclo.
- `printf(...)`: Viene stampato un menu che permette all'utente di selezionare uno degli esami disponibili (numerati da `1` a `6`), oppure l'opzione `0` per tornare indietro al menu principale.

Acquisizione dell'input dell'utente

```c
if (scanf("%d", &scelta) != 1) {
    printf("Input non valido.\n");
    while (getchar() != '\n'); // Pulisce il buffer di input
    continue;
}
```

- `scanf("%d", &scelta)`: Legge un numero intero inserito dall'utente e lo memorizza in `scelta`.
- **Verifica dell'input**: Se l'input non è un numero intero valido (ad esempio, se l'utente inserisce una stringa non numerica), `scanf` restituisce un valore diverso da `1`. In questo caso:
  - Viene stampato il messaggio `"Input non valido."`.
  - `while (getchar() != '\n');`: Pulisce il buffer di input, eliminando eventuali caratteri residui.
- `continue`: Riavvia il ciclo `while`, riproponendo il menu.

Gestione delle opzioni dell'utente

```
switch (scelta) {
    case 1:
        handle_exam_date("Reti di Calcolatori", "Reti", 3, should_return,
university_socket);
        break;
    case 2:
        handle_exam_date("Algoritmi e Strutture Dati", "Algoritmi", 3,
should_return, university_socket);
        break;
    case 3:
        handle_exam_date("Programmazione 1", "Prog1", 3, should_return,
university_socket);
        break;
    case 4:
        handle_exam_date("Programmazione 2", "Prog2", 3, should_return,
university_socket);
        break;
    case 5:
        handle_exam_date("Programmazione 3", "Prog3", 3, should_return,
university_socket);
        break;
    case 6:
        handle_exam_date("Tecnologie Web", "Web", 3, should_return,
university_socket);
        break;
    case 0:
        *should_return = 1; // Torna al menu principale
        return;
    default:
        system("clear");
        printf("Opzione non disponibile\n");
        *should_return = 1;
        return;
}
```

- `switch (scelta)`: Questa struttura `switch` decide l'azione da eseguire in base all'opzione scelta dall'utente.
  - `case 1 ... 6`: Per ciascun esame, viene chiamata la funzione `handle_exam_date`, che si occupa di:
    - Richiedere all'utente la data dell'esame.
    - Validare la data.
    - Formattare e inviare i dati al server tramite il socket `university_socket`.
    - Gli argomenti passati a `handle_exam_date` includono il nome dell'esame, il codice abbreviato, il tipo di esame (probabilmente `3` rappresenta un tipo di

esame specifico), e `should_return` per gestire il ritorno a eventuali menu principali.

- `case 0`: Se l'utente sceglie `0`, la funzione imposta `*should_return = 1` per segnalare che si desidera tornare al menu principale, quindi esce dalla funzione con `return`.
- `default`: Se l'utente inserisce un'opzione non valida (un numero che non rientra tra `0` e `6`), il programma:
  - Pulisce lo schermo.
  - Stampa `"Opzione non disponibile"`.
  - Imposta `*should_return = 1` per tornare al menu principale.
  - Esce dalla funzione.

Gestione del ritorno

```
if (should_return) {
    return; // Esci dalla funzione se necessario
}
```

- `if (should_return)`: Dopo che un'operazione è stata completata, viene verificato il valore di `should_return`. Se è stato impostato (cioè se è diverso da 0), la funzione esce con `return`.
  - Questo serve per consentire all'utente di tornare al menu principale dopo aver completato l'inserimento della data di un esame.

Funzione per gestire le richieste dei client

La funzione `handle_client` gestisce le richieste di un client connesso al server. Si occupa di creare una connessione con un server universitario (se non già presente) e di elaborare le comunicazioni del client. Di seguito è descritta in dettaglio ogni parte del codice.

```
void *handle_client(void *arg) {

    int client_socket = *(int *)arg;

    free(arg); // Libera la memoria allocata per il socket client

    char buffer[MAXLINE];

    int n;

    struct sockaddr_in university_addr;
```

```
    // Creazione del socket per la connessione all'università
    pthread_mutex_lock(&university_socket_mutex);
    if (university_socket <= 0) {
        if ((university_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
            perror("Errore nella creazione del socket università");
            pthread_mutex_unlock(&university_socket_mutex);
            close(client_socket);
            pthread_exit(NULL);
        }

        memset(&university_addr, 0, sizeof(university_addr));
        university_addr.sin_family = AF_INET;
        university_addr.sin_port = htons(12345);  // Porta del server universitario
        university_addr.sin_addr.s_addr = INADDR_ANY;

        if (connect(university_socket, (struct sockaddr *)&university_addr,
sizeof(university_addr)) < 0) {
            perror("Errore di connessione con l'università");
            close(university_socket);
            university_socket = -1;
            pthread_mutex_unlock(&university_socket_mutex);
            close(client_socket);
```

```
            pthread_exit(NULL);
        }
    }
    pthread_mutex_unlock(&university_socket_mutex);

    // Gestisce la comunicazione con il client
    while ((n = read(client_socket, buffer, MAXLINE - 1)) > 0) {
        buffer[n] = '\0';
        printf("Richiesta dal client: %s\n", buffer);

        // Invia la richiesta all'università
        pthread_mutex_lock(&university_socket_mutex);
        if (write(university_socket, buffer, strlen(buffer)) < 0) {
            perror("Errore in scrittura alla università");
            pthread_mutex_unlock(&university_socket_mutex);
            close(client_socket);
            pthread_exit(NULL);
        }

        // Riceve la risposta dall'università
        if ((n = read(university_socket, buffer, MAXLINE - 1)) <= 0) {
            perror("Errore nella lettura dalla università");
            pthread_mutex_unlock(&university_socket_mutex);
```

```
            close(client_socket);
            pthread_exit(NULL);
        }
        buffer[n] = '\0';
        pthread_mutex_unlock(&university_socket_mutex);

        // Invia la risposta dall'università allo studente
        if (write(client_socket, buffer, strlen(buffer)) < 0) {
            perror("Errore in scrittura allo studente");
            close(client_socket);
            pthread_exit(NULL);
        }
    }

    close(client_socket);
    pthread_exit(NULL);
}
```

Ricezione del socket del client e liberazione della memoria

```
int client_socket = *(int *)arg;
free(arg); // Libera la memoria allocata per il socket client
```

- `client_socket = *(int *)arg`: Il socket del client, passato come argomento `arg`, viene dereferenziato per ottenere l'intero corrispondente al file descriptor del socket.
- `free(arg)`: Libera la memoria precedentemente allocata per il socket del client. Questo è importante per evitare perdite di memoria.

Dichiarazione del buffer e altre variabili

```
char buffer[MAXLINE];
int n;
struct sockaddr_in university_addr;
```

- `buffer[MAXLINE]`: Buffer usato per memorizzare i dati ricevuti dal client o da inviare. `MAXLINE` è definito altrove e probabilmente rappresenta la dimensione massima del buffer (tipicamente 1024 byte).
- `n`: Variabile per memorizzare il numero di byte letti o scritti.
- `struct sockaddr_in university_addr`: Struttura che contiene le informazioni per connettersi al server universitario. Verrà usata per configurare l'indirizzo del server.

Blocco del mutex per l'accesso sicuro al socket dell'università

```
pthread_mutex_lock(&university_socket_mutex);
```

- `pthread_mutex_lock`: Blocca il mutex `university_socket_mutex`, garantendo che solo un thread alla volta possa accedere o modificare la risorsa condivisa, in questo caso `university_socket`. Questo è fondamentale perché più thread potrebbero cercare di connettersi contemporaneamente al server universitario.

Creazione del socket universitario (se non già esistente)

```
if (university_socket <= 0) {
    if ((university_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Errore nella creazione del socket università");
        pthread_mutex_unlock(&university_socket_mutex);
        close(client_socket);
        pthread_exit(NULL);
    }
```

- `if (university_socket <= 0)`: Verifica se il socket dell'università non è ancora stato creato o è stato chiuso (valori negativi o zero). Se il socket non esiste, deve essere creato.
- `socket(AF_INET, SOCK_STREAM, 0)`: Crea un nuovo socket:
  - `AF_INET`: Usa l'indirizzo IPv4.
  - `SOCK_STREAM`: Indica che il socket è di tipo TCP (stream).
  - `0`: Protocollo standard per TCP.
  - Se la creazione del socket fallisce (restituisce un valore negativo), stampa un messaggio d'errore e chiude il socket del client (`close(client_socket)`), sbloccando

il mutex e terminando il thread.

Configurazione dell'indirizzo del server universitario

```
memset(&university_addr, 0, sizeof(university_addr));
university_addr.sin_family = AF_INET;
university_addr.sin_port = htons(12345);  // Porta del server universitario
university_addr.sin_addr.s_addr = INADDR_ANY;
```

- `memset(&university_addr, 0, sizeof(university_addr))` : Inizializza a zero la struttura `university_addr`.
- `university_addr.sin_family = AF_INET` : Specifica che verrà utilizzato l'indirizzo di famiglia IPv4.
- `university_addr.sin_port = htons(12345)` : Imposta il numero di porta del server universitario a `12345`. La funzione `htons` (Host To Network Short) converte il numero di porta da formato host a formato di rete.
- `university_addr.sin_addr.s_addr = INADDR_ANY` : Indica che il server può essere su qualsiasi indirizzo locale disponibile.

Connessione al server universitario

```
if (connect(university_socket, (struct sockaddr *)&university_addr,
sizeof(university_addr)) < 0) {
    perror("Errore di connessione con l'università");
    close(university_socket);
    university_socket = -1;
    pthread_mutex_unlock(&university_socket_mutex);
    close(client_socket);
    pthread_exit(NULL);
}
```

- `connect(...)` : Tenta di connettersi al server universitario utilizzando il socket `university_socket`. Se la connessione fallisce (ritorna un valore negativo):
  - Viene stampato un messaggio d'errore.
  - `close(university_socket)` : Il socket creato viene chiuso per liberare risorse.
  - `university_socket = -1` : Il socket viene impostato a `-1` per indicare che non è valido.
  - `pthread_mutex_unlock(&university_socket_mutex)` : Viene sbloccato il mutex per consentire ad altri thread di accedere alla risorsa condivisa.
  - `close(client_socket)` : Il socket del client viene chiuso.
  - `pthread_exit(NULL)` : Il thread corrente termina la sua esecuzione.

Sblocco del mutex

```
pthread_mutex_unlock(&university_socket_mutex);
```

- `pthread_mutex_unlock(&university_socket_mutex)`: Se la connessione al server universitario ha avuto successo o se il socket era già creato, il mutex viene sbloccato per consentire ad altri thread di accedere.

Gestisce la comunicazione con il client

Codice che gestisce la comunicazione tra un client e un'università (tramite un socket), con l'uso di un mutex per sincronizzare l'accesso alla comunicazione con il server dell'università:

Inizio del ciclo di lettura dal client:

```
while ((n = read(client_socket, buffer, MAXLINE - 1)) > 0) {
```

- Il codice entra in un ciclo `while` che legge continuamente dati dal `client_socket` (socket collegato al client).
- `read`: legge fino a `MAXLINE - 1` byte dal socket del client e li memorizza in `buffer`. Restituisce il numero di byte letti (`n`). Se `n` è positivo, significa che c'è stato un input.
- Il ciclo continua finché ci sono dati da leggere dal client (ovvero finché `n` è maggiore di 0).

Termina la stringa letta:

```
buffer[n] = '\0';
```

- Dopo la lettura, la stringa viene terminata con `'\0'` per renderla una stringa C valida. Questo è necessario per evitare errori nella gestione delle stringhe.

Stampa della richiesta del client:

```
printf("Richiesta dal client: %s\n", buffer);
```

- La richiesta letta dal client viene stampata su standard output. Questo è utile per il debug o per tenere traccia delle richieste.

Blocco del mutex per la comunicazione con l'università:

```
pthread_mutex_lock(&university_socket_mutex);
```

- Si blocca il **mutex** (mutua esclusione) associato al socket dell'università. Questo garantisce che solo un thread alla volta possa accedere al socket dell'università, evitando problemi di concorrenza.

Invio della richiesta all'università:

```
if (write(university_socket, buffer, strlen(buffer)) < 0) {
    perror("Errore in scrittura alla università");
    pthread_mutex_unlock(&university_socket_mutex);
    close(client_socket);
    pthread_exit(NULL);
}
```

- Si invia il contenuto del `buffer` al server dell'università tramite il socket `university_socket`.
- `write`: invia la richiesta al server dell'università. Se l'operazione fallisce (ritorna un valore negativo), viene stampato un errore e:
  - Si sblocca il **mutex** per consentire ad altri thread di accedere.
  - Si chiude la connessione con il client.
  - Il thread termina la sua esecuzione con `pthread_exit(NULL)`.

Ricezione della risposta dall'università:

```
if ((n = read(university_socket, buffer, MAXLINE - 1)) <= 0) {
    perror("Errore nella lettura dalla università");
    pthread_mutex_unlock(&university_socket_mutex);
    close(client_socket);
    pthread_exit(NULL);
}
```

- Il codice legge la risposta dal server dell'università utilizzando `read` sul socket dell'università. La risposta viene memorizzata nuovamente in `buffer`.
- Se la lettura fallisce (`n <= 0`), viene stampato un errore, il mutex viene sbloccato e il thread si termina, come spiegato prima.

Termina la stringa della risposta:

```
buffer[n] = '\0';
```

- La risposta ricevuta viene terminata con `'\0'` per renderla una stringa C valida.

Sblocco del mutex:

```
pthread_mutex_unlock(&university_socket_mutex);
```

- Una volta ricevuta la risposta dall'università, il **mutex** viene sbloccato per consentire ad altri thread di inviare richieste all'università.

Invio della risposta al client:

```
if (write(client_socket, buffer, strlen(buffer)) < 0) {
    perror("Errore in scrittura allo studente");
    close(client_socket);
    pthread_exit(NULL);
}
```

- La risposta ricevuta dall'università viene inviata al client utilizzando `write`. Se l'invio fallisce, viene gestito con la chiusura del socket e la terminazione del thread.

Fine del ciclo e chiusura del socket del client:

```
close(client_socket);
pthread_exit(NULL);
```

- Una volta che non ci sono più dati da leggere dal client (quando il ciclo termina), si chiude il socket del client e il thread termina la sua esecuzione.

Funzione per gestire l'interazione terminale per aggiungere esami

Descrizione dettagliata del codice della funzione `manage_exams`, che gestisce l'interazione con l'utente tramite il terminale per aggiungere esami e gestire la connessione con un server universitario.

```
void *manage_exams() {

    system("clear");

    while (keep_running) {

        printf("Scegliere l'opzione:\n");

        printf("1) Aggiungi Esame\n");
```

```
        printf("0) Esci\n");


        int scelta;
        if (scanf("%d", &scelta) != 1) {
            printf("Input non valido.\n");
            while (getchar() != '\n'); // Pulisce il buffer di input
            continue;
        }


        while (getchar() != '\n'); // Pulisce il buffer di input


        switch (scelta) {
            case 1:
                pthread_mutex_lock(&university_socket_mutex);
                if (university_socket <= 0) {
                    struct sockaddr_in university_addr;
                    // Creazione del socket per la connessione all'università
                    if ((university_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
                        perror("Errore nella creazione del socket università");
                        pthread_mutex_unlock(&university_socket_mutex);
                        continue;
                    }
```

```
                    memset(&university_addr, 0, sizeof(university_addr));
                    university_addr.sin_family = AF_INET;
                    university_addr.sin_port = htons(12345);  // Porta del server universitario
                    university_addr.sin_addr.s_addr = INADDR_ANY;

                    if (connect(university_socket, (struct sockaddr *)&university_addr, sizeof(university_addr)) < 0) {
                        perror("Errore di connessione con l'università");
                        close(university_socket);
                        university_socket = -1;
                        pthread_mutex_unlock(&university_socket_mutex);
                        continue;
                    }
                }

                int should_return = 0;
                Inserisci_esame(university_socket, &should_return);
                if (should_return) {
                    pthread_mutex_unlock(&university_socket_mutex);
                    continue;
                }
                break;
```

```
            case 0:
                keep_running = 0; // Imposta la variabile globale per fermare il
ciclo
                pthread_mutex_lock(&university_socket_mutex);
                if (university_socket >= 0) {
                    close(university_socket);
                    university_socket = -1;
                }
                pthread_mutex_unlock(&university_socket_mutex);
                printf("Uscita dal programma.\n");
                // Chiudi il socket del server per uscire dal ciclo di accettazione
                if (server_socket >= 0) {
                    close(server_socket);
                    server_socket = -1;
                }
                // Esci dal thread
                pthread_exit(NULL);
                break;

            default:
                printf("Opzione non valida.\n");
                break;
        }
```

```
    }


    pthread_exit(NULL);
}
```

Pulizia del terminale all'inizio della funzione:

```
system("clear");
```

- Viene eseguito il comando `clear` per pulire il terminale, rendendo più chiaro all'utente l'interfaccia quando la funzione inizia.

Inizio del ciclo principale:

```
while (keep_running) {
```

- La funzione entra in un ciclo **while** che continua fino a quando la variabile globale `keep_running` rimane impostata su un valore diverso da zero. Questa variabile permette di controllare quando interrompere il ciclo, per esempio, quando l'utente sceglie di uscire.

Stampa del menu di scelta:

```
printf("Scegliere l'opzione:\n");
printf("1) Aggiungi Esame\n");
printf("0) Esci\n");
```

- Viene presentato un menu con due opzioni:
  - **1)** per aggiungere un esame.
  - **0)** per uscire dal programma.
  Acquisizione dell'input dell'utente:

```
int scelta;
if (scanf("%d", &scelta) != 1) {
    printf("Input non valido.\n");
    while (getchar() != '\n'); // Pulisce il buffer di input
```

```
    continue;
}
```

- La funzione legge l'input dell'utente utilizzando `scanf` per ottenere un numero intero che rappresenta la scelta.
- Se il valore inserito non è un intero (ad esempio, un carattere non valido), il programma notifica l'errore e pulisce il buffer di input per gestire eventuali caratteri residui, poi ricomincia il ciclo.

Pulizia del buffer di input:

```
while (getchar() != '\n'); // Pulisce il buffer di input
```

- Dopo aver letto l'input, il buffer viene svuotato per evitare che caratteri indesiderati rimangano nel buffer di input e interferiscano con future letture.

Gestione delle opzioni con uno switch:

```
switch (scelta) {
```

- A seconda dell'opzione scelta dall'utente, il programma esegue diverse operazioni.

Caso 1: Aggiungi Esame

Blocco del mutex per la connessione all'università:

```
pthread_mutex_lock(&university_socket_mutex);
```

- Viene bloccato un mutex per assicurarsi che l'accesso al socket dell'università sia sicuro (evitando accessi concorrenti da parte di altri thread).

Controllo della connessione esistente:

```
if (university_socket <= 0) {
```

- Se il socket dell'università (`university_socket`) è negativo o nullo (nessuna connessione attiva), il programma procede a creare una nuova connessione con il server universitario.

Creazione del socket per la connessione all'università:

```c
if ((university_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("Errore nella creazione del socket università");
    pthread_mutex_unlock(&university_socket_mutex);
    continue;
}
```

- Viene creato un socket TCP/IP usando `socket`. Se la creazione fallisce, viene stampato un errore e si sblocca il mutex, riprendendo il ciclo principale.

Configurazione dei parametri del server universitario:
`c memset(&university_addr, 0, sizeof(university_addr)); university_addr.sin_family = AF_INET; university_addr.sin_port = htons(12345); university_addr.sin_addr.s_addr = INADDR_ANY;`
- La struttura `sockaddr_in` viene inizializzata per definire il server universitario:
- `sin_family`: AF_INET per il protocollo IPv4.
- `sin_port`: La porta del server universitario viene impostata a `12345` (in rete, gli interi devono essere convertiti in formato big-endian con `htons`).
- `sin_addr.s_addr`: Impostato a `INADDR_ANY` per accettare connessioni su qualsiasi indirizzo locale disponibile.

Connessione al server universitario:
`c if (connect(university_socket, (struct sockaddr *)&university_addr, sizeof(university_addr)) < 0) { perror("Errore di connessione con l'università"); close(university_socket); university_socket = -1; pthread_mutex_unlock(&university_socket_mutex); continue; }`
- Si tenta di stabilire la connessione con l'università tramite `connect`. Se fallisce, viene chiuso il socket, ripristinata la variabile `university_socket` a -1, sbloccato il mutex e si riprende il ciclo.

Aggiunta dell'esame:
`c Inserisci_esame(university_socket, &should_return); if (should_return) { pthread_mutex_unlock(&university_socket_mutex); continue; }`
- Si chiama una funzione esterna `Inserisci_esame` (presumibilmente gestisce l'interazione con il server universitario per l'inserimento dell'esame). Se la funzione segnala che si deve interrompere (`should_return` è vero), si sblocca il mutex e si riprende il ciclo.

Caso 0: Uscita

Impostazione della variabile per uscire:
`c keep_running = 0;`
- Si imposta `keep_running` a 0 per uscire dal ciclo principale.

Chiusura della connessione con l'università:

`c pthread_mutex_lock(&university_socket_mutex); if (university_socket >= 0) { close(university_socket); university_socket = -1; } pthread_mutex_unlock(&university_socket_mutex);`

- Se la connessione con l'università è ancora attiva, il socket viene chiuso e la variabile `university_socket` viene reimpostata a -1 per segnalare che non c'è più una connessione.

Chiusura del server socket:

`c if (server_socket >= 0) { close(server_socket); server_socket = -1; }`

- Anche il socket del server viene chiuso se è ancora attivo.

Terminazione del thread:

`c pthread_exit(NULL);`

- Infine, il thread termina la sua esecuzione usando `pthread_exit`.

Caso Default: Opzione non valida

Gestione di input non validi:

`c printf("Opzione non valida.\n");`

- Se l'utente inserisce un'opzione che non corrisponde a 1 o 0, viene visualizzato un messaggio di errore.

MAIN

Descrizione dettagliata del codice della funzione `main()`, che gestisce un server multi-threaded in grado di accettare connessioni da client e un thread separato per gestire l'inserimento di esami:

```c
int main() {

    struct sockaddr_in server_addr, client_addr;

    socklen_t client_len = sizeof(client_addr);

    pthread_t thread_id, exam_thread;



    // Creazione del socket del server

    server_socket = socket(AF_INET, SOCK_STREAM, 0);

    if (server_socket < 0) {
```

```
        perror("Errore nella creazione del socket");
        exit(1);
    }

    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    if (bind(server_socket, (struct sockaddr *)&server_addr, sizeof(server_addr)) <
0) {
        perror("Errore nel bind");
        close(server_socket);
        exit(1);
    }

    if (listen(server_socket, 10) < 0) {
        perror("Errore nel listen");
        close(server_socket);
        exit(1);
    }

    printf("Segreteria in ascolto sulla porta %d...\n", PORT);
```

```
    // Crea il thread per gestire l'inserimento degli esami
    if (pthread_create(&exam_thread, NULL, manage_exams, NULL) != 0) {
        perror("Errore nella creazione del thread per l'inserimento degli esami");
        close(server_socket);
        exit(1);
    }
    pthread_detach(exam_thread);

    // Ciclo principale per accettare e gestire i client
    while (keep_running) {
        int client_socket = accept(server_socket, (struct sockaddr *)&client_addr,
&client_len);
        if (client_socket < 0) {
            if (!keep_running) break; // Se il server deve terminare, esci dal
ciclo
            perror("Errore nell'accept");
            continue;
        }

        int *client_sock_ptr = malloc(sizeof(int));
        if (client_sock_ptr == NULL) {
            perror("Errore nell'allocazione della memoria");
            close(client_socket);
            continue;
```

```
        }

        *client_sock_ptr = client_socket;


        // Crea un thread per gestire il client

        pthread_t client_thread;

        if (pthread_create(&client_thread, NULL, handle_client, client_sock_ptr) !=
0) {

            perror("Errore nella creazione del thread");

            close(client_socket);

            free(client_sock_ptr);

            continue;

        }

        pthread_detach(client_thread);

    }


    // Chiude il socket del server e aspetta la terminazione dei thread

    if (server_socket >= 0) {

        close(server_socket);

    }


    printf("Programma terminato.\n");


    return 0;
```

```
}
```

Dichiarazione delle strutture necessarie per la connessione:

```
struct sockaddr_in server_addr, client_addr;
socklen_t client_len = sizeof(client_addr);
pthread_t thread_id, exam_thread;
```

- `server_addr` e `client_addr` sono strutture che contengono informazioni sugli indirizzi IP e le porte per il server e il client, rispettivamente.
- `client_len` rappresenta la lunghezza della struttura `client_addr`.
- `pthread_t` sono identificatori per i thread creati nel programma.

Creazione del socket del server:

```
server_socket = socket(AF_INET, SOCK_STREAM, 0);
if (server_socket < 0) {
    perror("Errore nella creazione del socket");
    exit(1);
}
```

- Il socket del server viene creato usando la funzione `socket`, con il protocollo **AF_INET** (IPv4) e **SOCK_STREAM** (TCP).
- Se la creazione del socket fallisce (valore di ritorno negativo), il programma stampa un messaggio di errore e termina.

Inizializzazione della struttura `server_addr`:

```
memset(&server_addr, 0, sizeof(server_addr));
server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = INADDR_ANY;
server_addr.sin_port = htons(PORT);
```

- `memset` azzera la struttura `server_addr`.
- `sin_family` viene impostato su **AF_INET** per indicare il protocollo IPv4.
- `sin_addr.s_addr` viene impostato su **INADDR_ANY**, che significa che il server accetterà connessioni da qualsiasi interfaccia di rete locale.
- `sin_port` viene impostato su una porta specifica, convertita in formato big-endian tramite `htons(PORT)`.

Associazione del socket del server a un indirizzo IP e una porta:

```
if (bind(server_socket, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0)
{
    perror("Errore nel bind");
    close(server_socket);
    exit(1);
}
```

- `bind` collega il socket a un indirizzo e a una porta specificati in `server_addr`.
- Se l'associazione fallisce, viene stampato un messaggio di errore, il socket viene chiuso, e il programma termina.

Ascolto per connessioni in entrata:

```
if (listen(server_socket, 10) < 0) {
    perror("Errore nel listen");
    close(server_socket);
    exit(1);
}
```

- `listen` pone il socket del server in modalità di ascolto per le connessioni in entrata. Il secondo parametro specifica il numero massimo di connessioni in coda (qui 10).
- Se fallisce, il programma stampa un errore, chiude il socket e termina.

Messaggio di avvio del server:

```
printf("Segreteria in ascolto sulla porta %d...\n", PORT);
```

- Viene stampato un messaggio per indicare che il server è in ascolto sulla porta specificata.

Creazione del thread per la gestione degli esami:

```
if (pthread_create(&exam_thread, NULL, manage_exams, NULL) != 0) {
    perror("Errore nella creazione del thread per l'inserimento degli esami");
    close(server_socket);
    exit(1);
}
pthread_detach(exam_thread);
```

- Viene creato un thread separato utilizzando `pthread_create` per eseguire la funzione `manage_exams`, che gestisce l'inserimento degli esami. Se la creazione del thread fallisce,

viene stampato un errore e il server termina.

- `pthread_detach` viene chiamato per evitare che il thread debba essere "joinato" esplicitamente, consentendo la sua esecuzione indipendente.

Ciclo principale per accettare connessioni dai client:

```
while (keep_running) {
    int client_socket = accept(server_socket, (struct sockaddr *)&client_addr,
&client_len);
    if (client_socket < 0) {
        if (!keep_running) break;
        perror("Errore nell'accept");
        continue;
    }
```

- Il server entra in un ciclo `while` continuo, dove accetta connessioni dai client tramite `accept`.
- `accept` restituisce un nuovo socket per ogni client che si connette. Se fallisce, viene stampato un errore, e il ciclo continua, a meno che `keep_running` non sia impostato su 0, nel qual caso il ciclo si interrompe.

Allocazione dinamica della memoria per il socket del client:

```
int *client_sock_ptr = malloc(sizeof(int));
if (client_sock_ptr == NULL) {
    perror("Errore nell'allocazione della memoria");
    close(client_socket);
    continue;
}
*client_sock_ptr = client_socket;
```

- Viene allocata memoria dinamica per memorizzare il valore del `client_socket`. Questo è necessario perché il socket del client viene passato a un thread, e poiché i thread richiedono puntatori ai dati, è necessario allocare dinamicamente lo spazio per il socket.
- Se l'allocazione fallisce, viene stampato un errore e il ciclo riprende dopo aver chiuso la connessione con il client.

Creazione del thread per gestire il client:
`c pthread_t client_thread; if (pthread_create(&client_thread, NULL, handle_client, client_sock_ptr) != 0) { perror("Errore nella creazione del thread"); close(client_socket); free(client_sock_ptr); continue; } pthread_detach(client_thread);`
- Un nuovo thread viene creato per gestire ogni client che si connette, utilizzando la funzione

`handle_client`.
- Se la creazione del thread fallisce, viene stampato un errore, il socket del client viene chiuso e la memoria allocata viene liberata. Il ciclo riprende con la prossima connessione.
- `pthread_detach` viene chiamato per evitare che il thread debba essere unito esplicitamente.

Chiusura del socket del server:
`c if (server_socket >= 0) { close(server_socket); }`
- Quando il ciclo si interrompe, il socket del server viene chiuso se è ancora aperto.

Messaggio di terminazione del programma:
`c printf("Programma terminato.\n");`
- Un messaggio viene stampato per indicare che il programma è terminato.

Ritorno dal `main()`:
`c return 0;`
- Il programma termina con successo restituendo **0**.

Inclusione delle librerie necessarie

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

- `sys/types.h`, `sys/socket.h`, `netinet/in.h`: Queste librerie sono necessarie per l'utilizzo delle funzionalità di rete (come socket) per la comunicazione client-server.
- `stdio.h`: Fornisce funzionalità di input/output, come la stampa con `printf` o la lettura con `scanf`.
- `stdlib.h`: Contiene funzioni di utilità generale, come `malloc`, `free`, e `exit`.
- `string.h`: Fornisce funzioni per manipolare le stringhe, come `strcpy`, `strlen`, etc.
- `unistd.h`: Include funzioni per la gestione dei file, processi, e altro, come `read`, `write`, `close`, etc.

Definizioni delle costanti

```
#define PORT 12345              // Porta su cui il server universitario ascolta
#define MAXLINE 1024            // Dimensione massima del buffer
#define MAX_NAME_LENGTH 256     // Lunghezza massima per ID e nome dell'esame
#define MAX_RESERVATION 256     // Dimensione massima per le prenotazioni
#define MAX_STUDENTS 100        // Numero massimo di studenti
```

- `PORT`: Numero di porta su cui il server universitario ascolta le connessioni.
- `MAXLINE`: Dimensione massima del buffer per leggere e scrivere dati. Utilizzato per memorizzare i messaggi durante la comunicazione tra client e server.
- `MAX_NAME_LENGTH`: Lunghezza massima consentita per il nome di un esame o l'ID dell'esame (256 caratteri).
- `MAX_RESERVATION`: Numero massimo di prenotazioni che il sistema può gestire.
- `MAX_STUDENTS`: Numero massimo di studenti che possono registrarsi o fare prenotazioni.

Struttura per memorizzare informazioni sugli esami

```c
typedef struct {
    char id[MAX_NAME_LENGTH];     // ID dell'esame
    char name[MAX_NAME_LENGTH];   // Nome dell'esame
    char date[MAX_NAME_LENGTH];   // Data dell'esame
    int num_exam;                 // Numero totale degli esami
} Exam;
```

- `Exam`: Definisce una struttura che rappresenta un esame. Contiene:
  - `id`: Identificatore univoco dell'esame (es. "Prog1" per Programmazione 1).
  - `name`: Nome completo dell'esame (es. "Programmazione 1").
  - `date`: Data in cui si svolgerà l'esame (es. "1-10-2024").
  - `num_exam`: Numero totale degli esami disponibili. Questo campo viene usato per gestire la lista degli esami.

Struttura per memorizzare le prenotazioni degli esami

```c
typedef struct {
    int student_id;               // ID dello studente
    Exam exam;                    // Informazioni sull'esame prenotato
    int num_prenotazioni;         // Numero di prenotazioni per questo esame
} Prenotazioni;
```

- `Prenotazioni`: Definisce una struttura per rappresentare una prenotazione d'esame. Contiene:
  - `student_id`: ID univoco dello studente che ha prenotato l'esame.
  - `exam`: Informazioni sull'esame prenotato (usa la struttura `Exam` definita prima).
  - `num_prenotazioni`: Numero di prenotazioni fatte dallo studente per questo esame. Può essere utile per limitare il numero di posti disponibili o evitare doppie prenotazioni.

Funzione per inizializzare gli esami con dati predefiniti

```c
void init_exam(Exam *exam) {
    // Inizializzazione degli esami
    strcpy(exam[0].id, "Reti");
    strcpy(exam[0].name, "Reti di Calcolatori");
    strcpy(exam[0].date, "9-10-2024");

    strcpy(exam[1].id, "Algoritmi");
    strcpy(exam[1].name, "Algoritmi e Strutture Dati");
    strcpy(exam[1].date, "15-9-2024");
```

```
    strcpy(exam[2].id, "Prog1");
    strcpy(exam[2].name, "Programmazione 1");
    strcpy(exam[2].date, "1-10-2024");

    strcpy(exam[3].id, "Reti");
    strcpy(exam[3].name, "Reti di Calcolatori");
    strcpy(exam[3].date, "20-8-2024");

    strcpy(exam[4].id, "Prog2");
    strcpy(exam[4].name, "Programmazione 2");
    strcpy(exam[4].date, "10-10-2024");

    strcpy(exam[5].id, "Prog3");
    strcpy(exam[5].name, "Programmazione 3");
    strcpy(exam[5].date, "1-11-2024");

    strcpy(exam[6].id, "Web");
    strcpy(exam[6].name, "Tecnologie Web");
    strcpy(exam[6].date, "1-9-2024");

    exam->num_exam = 7; // Imposta il numero totale degli esami
}
```

- `init_exam`: Funzione che inizializza una lista di esami con dati predefiniti.
  - `exam`: Un array di strutture `Exam` viene passato per riferimento. La funzione inserisce informazioni statiche in ogni elemento dell'array, come l'ID, il nome e la data dell'esame.
  - `strcpy`: Copia le stringhe nelle variabili della struttura.
    - `exam[0]` rappresenta "Reti di Calcolatori", con una data del **9-10-2024**.
    - `exam[1]` rappresenta "Algoritmi e Strutture Dati", e così via per altri esami.
  - Alla fine, imposta `exam->num_exam` a 7, indicando il numero totale di esami disponibili nel sistema.

Richiesta Prenotazione

```
void Richiesta_Prenotazione(char *buffer, char *response, Exam exam[], Prenotazioni prenotazioni[]) {
```

- `buffer`: Contiene la richiesta del client, che può essere la richiesta di mostrare esami disponibili o una prenotazione effettiva.

- `response`: Buffer in cui verrà memorizzata la risposta del server, come il risultato della richiesta di prenotazione.
- `exam[]`: Un array di strutture `Exam` che contiene le informazioni sugli esami disponibili (id, nome, data, ecc.).
- `prenotazioni[]`: Un array di strutture `Prenotazioni` che memorizza le prenotazioni effettuate dagli studenti.

Variabili locali

```
int id_studente;
int scelta;
int count = 0;
```

- `id_studente`: ID dello studente che sta facendo la prenotazione.
- `scelta`: Rappresenta l'indice dell'esame che lo studente vuole prenotare.
- `count`: Contatore per tracciare il numero di esami disponibili durante la visualizzazione delle date.

Pulizia dello schermo

```
system("clear"); // Pulisce lo schermo del terminale
```

- Viene eseguito il comando `clear` per pulire lo schermo del terminale, utile per migliorare la leggibilità e l'interfaccia per l'utente.

Controllo del tipo di richiesta

```
if (strcmp(buffer, "MostraEsami_2") == 0) {
```

- Controlla se il buffer contiene la stringa `MostraEsami_2`, che indica una richiesta per mostrare tutte le date disponibili degli esami.

Mostra tutte le date degli esami disponibili

```
char temp[MAXLINE] = "Date Disponibili:\n";
for (int i = 0; i < exam->num_exam; i++) {
    printf("%d)%s, Data: %s\n", i+1, exam[i].name, exam[i].date);
    snprintf(temp + strlen(temp), MAXLINE - strlen(temp), "%d)%s, Data: %s\n", i +
1, exam[i].name, exam[i].date);
    count++;
```

```
}
snprintf(response, MAXLINE, "Numero di esami: %d\n%s", count, temp);
```

- In questo blocco:
  - Viene inizializzata una stringa temporanea `temp` che conterrà tutte le informazioni sulle date degli esami.
  - Il ciclo `for` scorre attraverso tutti gli esami nell'array `exam[]`, stampa e accumula le informazioni sugli esami (nome e data) sia sul terminale che in `temp`.
  - `count` tiene traccia del numero di esami disponibili.
  - Alla fine del ciclo, la variabile `response` viene riempita con il numero totale di esami disponibili e la lista delle date, pronta per essere inviata come risposta al client.

Estrarre l'ID dello studente e la scelta dell'esame

```
sscanf(buffer, "%d:%d_2", &id_studente, &scelta);
```

- Se la richiesta non è quella di mostrare gli esami, allora si tratta di una prenotazione. La funzione `sscanf` estrae l'ID dello studente e l'indice dell'esame (scelta) dal buffer.
  - `id_studente`: ID dello studente che sta prenotando.
  - `scelta`: Indice dell'esame selezionato dallo studente (in base all'elenco degli esami mostrato in precedenza).

Controllo se l'esame è già prenotato dallo studente

```
for (size_t i = 0; i < prenotazioni->num_prenotazioni; i++) {
    if ((prenotazioni[i].student_id == id_studente) &&
        strcmp(prenotazioni[i].exam.id, exam[scelta].id) == 0 &&
        strcmp(prenotazioni[i].exam.date, exam[scelta].date) == 0) {
        snprintf(response, MAXLINE, "Prenotazione Giá Effettuata");
        return;
    }
}
```

- Questo ciclo `for` controlla se lo studente ha già prenotato l'esame selezionato.
  - Per ogni prenotazione, verifica se l'**ID dello studente**, l'**ID dell'esame**, e la **data dell'esame** corrispondono alla prenotazione corrente.
  - Se una prenotazione esiste già, la risposta viene impostata su **"Prenotazione Giá Effettuata"**, e la funzione termina.

Aggiungere una nuova prenotazione

```
prenotazioni[prenotazioni->num_prenotazioni].student_id = id_studente;
prenotazioni[prenotazioni->num_prenotazioni].exam = exam[scelta];
prenotazioni->num_prenotazioni = prenotazioni->num_prenotazioni + 1;
```

- Se lo studente non ha già prenotato l'esame, viene aggiunta una nuova prenotazione:
  - `prenotazioni[prenotazioni->num_prenotazioni].student_id`: L'ID dello studente viene assegnato alla nuova prenotazione.
  - `prenotazioni[prenotazioni->num_prenotazioni].exam`: Viene assegnato l'esame scelto dallo studente.
  - Infine, `num_prenotazioni` viene incrementato di 1 per registrare la nuova prenotazione.

Mostra tutte le prenotazioni effettuate

```
for (int i = 0; i < prenotazioni->num_prenotazioni; i++) {
    printf("%d)ID Studente: %d, Esame: %s, Data: %s\n", i,
prenotazioni[i].student_id, prenotazioni[i].exam.name, prenotazioni[i].exam.date);
}
snprintf(response, MAXLINE, "Esame Prenotato. Numero Prenotazioni %d",
prenotazioni->num_prenotazioni);
```

- Dopo aver aggiunto la prenotazione, il programma stampa sul terminale tutte le prenotazioni effettuate finora.
- Viene costruita una risposta in `response` che conferma che l'esame è stato prenotato e include il numero totale di prenotazioni.

Funzione `Esami_Disponibili`

Questa funzione mostra le date disponibili per un determinato esame richiesto.

```
void Esami_Disponibili(char *buffer, char *response, Exam exam[]) {
    size_t len = strlen(buffer);
```

- `buffer`: Contiene l'ID dell'esame che l'utente ha richiesto.
- `response`: Buffer in cui la funzione memorizza la risposta da inviare al client.
- `exam[]`: Array di esami disponibili (di tipo `Exam`), che contiene ID, nome, data e altre informazioni.
- `strlen(buffer)`: Calcola la lunghezza della stringa contenuta nel buffer.

Estrazione dell'ID dell'esame

```
    char esame[MAX_NAME_LENGTH];
    strncpy(esame, buffer, len - 2); // Copia l'ID dell'esame, ignorando gli ultimi due caratteri ('\n' e '\0')
    esame[len - 2] = '\0';
```

- `esame[]`: Variabile che memorizza l'ID dell'esame.
- `strncpy(esame, buffer, len - 2)`: Copia i primi `len - 2` caratteri dal `buffer` in `esame`, rimuovendo gli ultimi due caratteri, che potrebbero essere `\n` o `\0`.
- `esame[len - 2] = '\0'`: Termina la stringa `esame` con il carattere nullo.
- `printf("ID Esame: %s\n", esame)`: Stampa l'ID dell'esame estratto per debugging.

Costruzione della risposta con le date disponibili

```
    char temp[MAXLINE] = "Date Disponibili:\n";

    for (int i = 0; i < exam->num_exam; i++) {
        if (strcmp(exam[i].id, esame) == 0) {
            printf("%d)%s, Data: %s\n", i, exam[i].name, exam[i].date);
            snprintf(temp + strlen(temp), MAXLINE - strlen(temp), "%d)%s, Data: %s\n", i, exam[i].name, exam[i].date);
        }
    }
```

- `temp[MAXLINE]`: Stringa temporanea che inizializza la risposta con il testo "Date Disponibili:\n".
- **Ciclo** `for`: Scorre tutti gli esami nell'array `exam[]`.
  - `strcmp(exam[i].id, esame)`: Confronta l'ID dell'esame corrente con quello estratto dal buffer.
  - Se l'ID corrisponde, la funzione stampa e aggiunge le informazioni dell'esame (nome e data) a `temp`.
  - `snprintf(temp + strlen(temp), ...)`: Appende le informazioni a `temp`, garantendo che la lunghezza totale non superi `MAXLINE`.

Creazione della risposta finale

```
    snprintf(response, MAXLINE, "%s", temp);
}
```

- Infine, la risposta viene memorizzata nel buffer `response` pronto per essere inviato al client. Contiene le date degli esami disponibili.

Funzione `aggiungi_esame`

Questa funzione permette di aggiungere un nuovo esame al sistema.

```
void aggiungi_esame(Exam *exam, int *num_exam, const char *id, const char *name,
const char *date, char *response) {
```

- `exam[]` : Array di esami in cui sarà aggiunto il nuovo esame.
- `num_exam` : Puntatore che tiene traccia del numero corrente di esami presenti nel sistema.
- `id` : ID dell'esame da aggiungere.
- `name` : Nome dell'esame da aggiungere.
- `date` : Data dell'esame da aggiungere.
- `response` : Buffer in cui memorizzare la risposta per il client (conferma o errore).

Controllo del limite massimo di esami

```
    if (*num_exam < MAX_RESERVATION) {
```

- Controlla se il numero di esami attuali (`*num_exam`) è inferiore al limite massimo definito da `MAX_RESERVATION` . Questo garantisce che non vengano superati i limiti del sistema per il numero di esami gestibili.

Aggiunta del nuovo esame

```
        strncpy(exam[*num_exam].id, id, MAX_NAME_LENGTH);
        strncpy(exam[*num_exam].name, name, MAX_NAME_LENGTH);
        strncpy(exam[*num_exam].date, date, MAX_NAME_LENGTH);
```

- Viene copiato l'**ID**, il **nome** e la **data** dell'esame nei campi della struttura `exam[]` al prossimo indice disponibile (`*num_exam`).
  - `strncpy` : Copia in modo sicuro fino a `MAX_NAME_LENGTH` caratteri dai parametri `id` , `name` e `date` nei rispettivi campi della struttura.

Conferma dell'aggiunta

```
        snprintf(response, MAXLINE, "SERVER) Appello aggiunto: ID=%s, Data=%s\n",
id, date);
        printf("Appello aggiunto: ID=%s, Data=%s\n", id, date);
```

- `snprintf`: Costruisce una stringa di conferma che indica che l'appello è stato aggiunto correttamente. Questa stringa viene memorizzata in `response`.
- `printf`: Stampa lo stesso messaggio di conferma sul terminale del server.

Incremento del contatore di esami

```
        (*num_exam)++;
```

- Incrementa il contatore degli esami, segnalando che un nuovo esame è stato aggiunto.

Gestione dell'errore (limite raggiunto)

```
    } else {
        snprintf(response, MAXLINE, "Impossibile aggiungere l'appello, limite
raggiunto.\n");
        printf("Impossibile aggiungere l'appello, limite raggiunto.\n");
    }
}
```

- Se il limite massimo di esami (`MAX_RESERVATION`) è stato raggiunto, la funzione genera un messaggio di errore sia nel `response` (per il client) che sul terminale del server.

Funzione gestione richiesta `Aggiungi_Esame`

Questa funzione gestisce la richiesta di aggiunta di un nuovo esame da parte del client.

```
void Aggiungi_Esame(char *buffer, char *response, Exam exam[]) {
    char id[MAX_NAME_LENGTH];
    char date[MAX_NAME_LENGTH];
    char suffix[MAXLINE];
```

- `buffer`: Stringa che contiene la richiesta del client, inclusi i dettagli dell'esame da aggiungere.
- `response`: Buffer che conterrà la risposta da inviare al client.

- `id`, `date`, `suffix`: Variabili che contengono rispettivamente l'ID dell'esame, la data e un suffisso usato per identificare il tipo di richiesta (in questo caso, il suffisso "3" indica l'aggiunta di un esame).

Estrazione dei dettagli dell'esame dal buffer

```
    if (sscanf(buffer, "%[^:]:%[^_]_%s", id, date, suffix) == 3 && strcmp(suffix,
"3") == 0) {
```

- `sscanf`: Estrae l'ID dell'esame, la data e il suffisso dalla stringa `buffer`. Utilizza il formato `"%[^:]:%[^_]_%s"` per dividere il buffer nei componenti separati da `:` e `_`.
- **Controllo**: Se il suffisso è "3", significa che la richiesta è valida e si procede con l'aggiunta di un esame.

Verifica dell'esistenza della data per l'esame

```
        for (int i = 0; i < exam->num_exam; i++) {
            if (strcmp(exam[i].id, id) == 0 && strcmp(exam[i].date, date) == 0) {
                snprintf(response, MAX_NAME_LENGTH, "Data %s per l'esame ID=%s già
esistente.\n", date, id);
                printf("%s", response);
                return;
            }
        }
```

- **Ciclo** `for`: Scorre tutti gli esami nell'array `exam[]`.
- **Controllo**: Se esiste già un esame con lo stesso ID e la stessa data, invia un messaggio di errore al client, indicando che la data per quell'esame è già registrata.

Definizione dei nomi degli esami

```
        const char *names[] = {
            "Reti di Calcolatori",
            "Algoritmi e Strutture Dati",
            "Programmazione 1",
            "Programmazione 2",
            "Programmazione 3",
            "Tecnologie Web"
        };
        const char *ids[] = {"Reti", "Algoritmi", "Prog1", "Prog2", "Prog3",
"Web"};
```

- `names[]` **e** `ids[]`: Array che contengono rispettivamente i nomi completi degli esami e i relativi ID. Questi array sono usati per validare l'ID dell'esame e associare l'ID a un nome.

Aggiunta del nuovo esame

```
        for (int i = 0; i < 6; i++) {
            if (strcmp(id, ids[i]) == 0) {
                aggiungi_esame(exam, &exam->num_exam, id, names[i], date,
response);
                return;
            }
        }
```

- **Ciclo** `for`: Scorre gli ID validi e, se l'ID corrisponde, chiama la funzione `aggiungi_esame` per aggiungere l'esame al sistema, inviando anche una conferma al client tramite il buffer `response`.

Stampa delle date disponibili

```
        char temp[MAXLINE] = "Date Disponibili:\n";
        for (int i = 0; i < exam->num_exam; i++) {
            printf("ID: %s, Name: %s, Data: %s\n", exam[i].id, exam[i].name,
exam[i].date);
        }
    }
}
```

- Se l'esame viene aggiunto correttamente, il programma stampa tutte le date degli esami disponibili per monitoraggio o debugging.

Funzione `handle_client`

Questa funzione gestisce la comunicazione con un singolo client e processa le varie richieste.

```
void handle_client(int client_socket, Exam exam[], Prenotazioni prenotazioni[]) {
    char buffer[MAXLINE];
    int n;
```

- `client_socket`: Socket del client per la comunicazione.
- `buffer[MAXLINE]`: Buffer per leggere i dati dal client.

- `exam[]`: Array di esami disponibili.
- `prenotazioni[]`: Array di prenotazioni fatte dagli studenti.

Ciclo per leggere i dati dal client

```
    while ((n = read(client_socket, buffer, MAXLINE - 1)) > 0) {
        buffer[n] = '\0'; // Aggiungi il terminatore di stringa
        printf("Received: %s\n", buffer);
```

- `read`: Legge i dati dal socket del client e li memorizza in `buffer`. Restituisce il numero di byte letti.
- `buffer[n] = '\0'`: Aggiunge un terminatore di stringa per rendere `buffer` una stringa C valida.
- `printf`: Stampa il messaggio ricevuto dal client per monitoraggio.

Identificazione della richiesta

```
        char last_char = buffer[strlen(buffer) - 1]; // Ottieni l'ultimo carattere
del buffer
        char response[MAXLINE];
```

- `last_char`: Ottiene l'ultimo carattere della stringa `buffer`, che viene utilizzato per identificare il tipo di richiesta inviata dal client (es. '1' per visualizzare esami, '2' per prenotare un esame, '3' per aggiungere un nuovo esame).
- `response`: Buffer in cui verrà memorizzata la risposta da inviare al client.

Gestione delle richieste del client

```
        if (last_char == '1') {
            Esami_Disponibili(buffer, response, exam); // Richiesta di esami
disponibili
        } else if (last_char == '2') {
            Richiesta_Prenotazione(buffer, response, exam, prenotazioni); //
Prenotazione esame
        } else if (last_char == '3') {
            Aggiungi_Esame(buffer, response, exam); // Aggiungi un nuovo esame
        }
```

- `if (last_char == '1')`: Se l'ultimo carattere della stringa ricevuta è '1', il client ha richiesto di visualizzare gli esami disponibili, quindi viene chiamata la funzione `Esami_Disponibili`.

- `else if (last_char == '2')`: Se l'ultimo carattere è '2', viene chiamata la funzione `Richiesta_Prenotazione` per gestire la prenotazione di un esame.
- `else if (last_char == '3')`: Se l'ultimo carattere è '3', il client ha richiesto di aggiungere un nuovo esame, quindi viene chiamata la funzione `Aggiungi_Esame`.

Invio della risposta al client

```
        if (FullWrite(client_socket, response, strlen(response)) < 0) {
            perror("Write error");
            close(client_socket);
            return;
        }
    }
    close(client_socket); // Chiudi la connessione con il client
}
```

- `FullWrite`: Questa è una funzione personalizzata (non standard) che invia tutti i byte della stringa `response` al client attraverso il socket `client_socket`. Il motivo per cui si usa una funzione specifica come `FullWrite` è che la funzione standard `write` non garantisce l'invio di tutti i byte richiesti in una sola chiamata.
  Valore di ritorno di `FullWrite`: Se l'invio dei dati ha successo, `FullWrite` ritorna il numero di byte effettivamente inviati. Se si verifica un errore durante l'invio, ritorna un valore negativo (solitamente -1).
- Se c'è un errore durante l'invio della risposta, viene stampato un messaggio di errore e il socket del client viene chiuso.
- Dopo aver terminato il ciclo di lettura e scrittura, la connessione con il client viene chiusa con `close(client_socket)`.

Main

```
int main() {

    int server_socket, client_socket;

    struct sockaddr_in server_addr, client_addr;

    socklen_t client_len = sizeof(client_addr);

    Prenotazioni prenotazioni[MAX_RESERVATION];
```

```
    prenotazioni->num_prenotazioni = 0; // Inizializza il numero di prenotazioni
    Exam exam[MAX_EXAM];
    init_exam(exam); // Inizializza gli esami


    // Creazione del socket del server
    server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket < 0) {
        perror("Socket creation error");
        exit(1);
    }


    // Configurazione dell'indirizzo del server
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);


    // Associa l'indirizzo al socket
    if (bind(server_socket, (struct sockaddr *)&server_addr, sizeof(server_addr)) <
0) {
        perror("Bind error");
        close(server_socket);
        exit(1);
    }
```

```
    // Metti il socket in ascolto delle connessioni in entrata
    if (listen(server_socket, 10) < 0) {
        perror("Listen error");
        close(server_socket);
        exit(1);
    }
    system("clear"); // Pulisce lo schermo
    printf("Server Universitario in ascolto sulla porta %d...\n", PORT);

    // Ciclo principale per accettare e gestire le connessioni dei client
    while (1) {
        client_socket = accept(server_socket, (struct sockaddr *)&client_addr,
&client_len);
        if (client_socket < 0) {
            perror("Accept error");
            return 0;
        }
        handle_client(client_socket, exam, prenotazioni); // Gestisci il client
    }

    close(server_socket); // Chiudi il socket del server
    return 0;
```

```
}
```

Descrizione:

```
int server_socket, client_socket;
struct sockaddr_in server_addr, client_addr;
socklen_t client_len = sizeof(client_addr);
Prenotazioni prenotazioni[MAX_RESERVATION];
prenotazioni->num_prenotazioni = 0; // Inizializza il numero di prenotazioni
Exam exam[7];
```

- `server_socket`: Variabile che memorizza il file descriptor del socket del server.
- `client_socket`: File descriptor del socket del client che si connette al server.
- `server_addr`: Struttura che memorizza l'indirizzo del server (indirizzo IP, porta, ecc.).
- `client_addr`: Struttura che memorizza l'indirizzo del client che si connette.
- `client_len`: Lunghezza della struttura `client_addr`, necessaria per la funzione `accept`.
- `prenotazioni[MAX_RESERVATION]`: Array di strutture `Prenotazioni` che memorizza le prenotazioni di esami fatte dagli studenti. `MAX_RESERVATION` è il numero massimo di prenotazioni gestibili.
- `exam[7]`: Array di strutture `Exam` che memorizza le informazioni sugli esami. In questo caso, sono disponibili 7 esami.

Inizializzazione delle prenotazioni e degli esami

```
prenotazioni->num_prenotazioni = 0; // Inizializza il numero di prenotazioni
init_exam(exam); // Inizializza gli esami
```

- `prenotazioni->num_prenotazioni`: Inizializza a 0 il numero di prenotazioni, per indicare che inizialmente non ci sono prenotazioni.
- `init_exam(exam)`: Funzione che riempie l'array `exam[]` con i dettagli predefiniti degli esami (ID, nome, data, ecc.).

Creazione del socket del server

```
server_socket = socket(AF_INET, SOCK_STREAM, 0);
if (server_socket < 0) {
    perror("Socket creation error");
    exit(1);
}
```

- `socket(AF_INET, SOCK_STREAM, 0)`: Crea un socket di tipo **TCP** (specificato da `SOCK_STREAM`) con protocollo IPv4 (`AF_INET`). Restituisce un file descriptor, che viene memorizzato in `server_socket`.
- **Controllo di errore**: Se la creazione del socket fallisce (`server_socket < 0`), il programma stampa un messaggio di errore e termina con `exit(1)`.

Configurazione dell'indirizzo del server

```
memset(&server_addr, 0, sizeof(server_addr));
server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = INADDR_ANY;
server_addr.sin_port = htons(PORT);
```

- `memset`: Inizializza la struttura `server_addr` a 0.
- `server_addr.sin_family = AF_INET`: Indica che il server utilizza il protocollo IPv4.
- `server_addr.sin_addr.s_addr = INADDR_ANY`: Il server accetta connessioni da qualsiasi interfaccia di rete disponibile.
- `server_addr.sin_port = htons(PORT)`: Imposta la porta su cui il server ascolta, convertita in formato di rete (big-endian) con `htons()`.

Associazione del socket all'indirizzo del server

```
if (bind(server_socket, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0)
{
    perror("Bind error");
    close(server_socket);
    exit(1);
}
```

- `bind()`: Associa il socket creato all'indirizzo specificato in `server_addr`. Questo collega il socket al protocollo IP e alla porta definiti.
- **Controllo di errore**: Se l'associazione fallisce, il programma stampa un messaggio di errore, chiude il socket e termina.

Messa in ascolto per connessioni in entrata

```
if (listen(server_socket, 10) < 0) {
    perror("Listen error");
    close(server_socket);
```

```
    exit(1);
}
```

- `listen(server_socket, 10)` : Pone il socket in modalità di ascolto, specificando che può accettare fino a 10 connessioni pendenti. Il socket diventa così passivo, in attesa di connessioni dai client.
- **Controllo di errore**: Se la funzione `listen()` fallisce, il programma stampa un errore, chiude il socket e termina.

Messaggio di conferma del server in ascolto

```
printf("Server Universitario in ascolto sulla porta %d...\n", PORT);
```

- Stampa un messaggio sul terminale per indicare che il server è correttamente in ascolto sulla porta specificata.

Ciclo principale per gestire le connessioni dei client

```
while (1) {
    client_socket = accept(server_socket, (struct sockaddr *)&client_addr,
&client_len);
    if (client_socket < 0) {
        perror("Accept error");
        return 0;
    }
    handle_client(client_socket, exam, prenotazioni); // Gestisci il client
}
```

- `while(1)` : Il server entra in un ciclo infinito per accettare continuamente connessioni dai client.
- `accept()` : Accetta una connessione in entrata da un client. Restituisce un nuovo file descriptor (`client_socket`), che rappresenta la connessione con il client.
    - `client_socket` : Socket specifico per il client connesso. Da questo punto in poi, il server può inviare e ricevere dati dal client tramite questo socket.
- **Controllo di errore**: Se la funzione `accept()` fallisce, viene stampato un errore e il ciclo si interrompe.
- `handle_client(client_socket, exam, prenotazioni)` : Viene chiamata la funzione `handle_client` per gestire la comunicazione con il client. Questa funzione si occupa di leggere la richiesta del client e rispondere in base alla richiesta (visualizzare esami, aggiungere prenotazioni, ecc.).

Chiusura del socket del server

```
close(server_socket);
return 0;
```

- `close(server_socket)` : Se il ciclo principale viene interrotto, il socket del server viene chiuso per rilasciare le risorse.
- `return 0` : Il programma termina con successo.

CODICE SORGENTE STUDENTE

Macro `max(x, y)`

```
#define max(x, y) ({typeof (x) x_ = (x); typeof (y) y_ = (y); x_ > y_ ? x_ : y_;})
```

- **Cos'è:** Una macro definita per trovare il massimo tra due valori `x` e `y`. Le macro in C vengono espanse dal preprocessore e possono essere viste come una scorciatoia.
- **Passaggi del codice:**
  1. `typeof(x)`: Si utilizza `typeof`, che è un'estensione GNU C, per ottenere il tipo di `x`. Ciò significa che il tipo di `x` verrà valutato automaticamente dal compilatore.
  2. `x_ = (x)` **e** `y_ = (y)`: La macro assegna il valore di `x` e `y` a due variabili temporanee `x_` e `y_`. Questo evita problemi quando `x` o `y` vengono usati in espressioni più complesse, garantendo che siano valutati solo una volta.
  3. `x_ > y_ ? x_ : y_`: È un operatore ternario che restituisce il valore più grande tra `x_` e `y_`. Se `x_` è maggiore di `y_`, restituisce `x_`, altrimenti restituisce `y_`.

Funzione `FullWrite`

```
ssize_t FullWrite(int fd, const void *buf, size_t count) {
    size_t nleft = count;
    ssize_t nwritten;
    const char *ptr = buf;

    while (nleft > 0) {
        if ((nwritten = write(fd, ptr, nleft)) <= 0) {
            if (nwritten < 0 && errno == EINTR) {
                continue;
            } else {
                return -1;
            }
        }
        nleft -= nwritten;
        ptr += nwritten;
    }
    return count;
}
```

Descrizione della funzione:

Questa funzione cerca di **scrivere completamente il contenuto** del buffer su un file descriptor specifico (tipicamente un file aperto o una socket), e si assicura di gestire eventuali errori di

sistema, come interruzioni di sistema.

- **Tipo di ritorno e parametri:**
  - `ssize_t`: Tipo di dato per rappresentare un risultato che può essere negativo in caso di errore, o positivo in caso di successo (come il numero di byte scritti).
  - `int fd`: Il file descriptor su cui si vuole scrivere. È un identificatore per file, socket o altri flussi di dati.
  - `const void *buf`: Il puntatore al buffer che contiene i dati da scrivere.
  - `size_t count`: Il numero di byte da scrivere dal buffer.

Flusso di esecuzione:

1. **Inizializzazione**:
   - `nleft` viene inizializzato a `count`, che rappresenta il numero totale di byte che devono essere scritti.
   - `ptr` è un puntatore al buffer, inizialmente puntato all'inizio dei dati.
2. **Ciclo `while`**:
   - Si entra in un ciclo che continuerà fino a quando non sono stati scritti tutti i byte (`nleft > 0`).
3. **Chiamata a `write()`**:
   - `nwritten = write(fd, ptr, nleft)`: La funzione `write()` tenta di scrivere fino a `nleft` byte dal buffer `ptr` nel file descriptor `fd`. Restituisce il numero di byte scritti o un valore negativo in caso di errore.
   - **Gestione degli errori**:
     - Se `write()` ritorna `<= 0`, significa che c'è stato un errore o l'operazione non ha scritto alcun byte.
     - **Caso speciale di errore `EINTR`**: Questo errore indica che l'operazione è stata interrotta da un segnale. In questo caso, si usa `continue` per riprovare la scrittura senza interrompere il processo.
     - **Altri errori**: Se l'errore non è `EINTR`, si restituisce `-1` indicando il fallimento.
4. **Aggiornamento dei byte rimanenti**:
   - `nleft -= nwritten`: Dopo ogni scrittura, il numero di byte rimasti da scrivere (`nleft`) viene aggiornato sottraendo il numero di byte appena scritti (`nwritten`).
   - **Aggiornamento del puntatore al buffer**: `ptr += nwritten` sposta il puntatore `ptr` in avanti di `nwritten` byte, per assicurarsi che la prossima chiamata a `write()` scriva la parte rimanente del buffer.
5. **Ritorno finale**:
   - Quando tutti i byte sono stati scritti (`nleft` è uguale a zero), la funzione restituisce `count`, ovvero il numero totale di byte scritti con successo.

La funzione si occupa di richiedere una lista di esami disponibili al server, leggere la risposta, permettere all'utente di selezionare un esame e gestire eventuali errori di input.

Funzione `Richiesta_Prenotazione`

```
void Richiesta_Prenotazione(int socket, int id_studente) {
    int scelta;
    int nwrite;
    char buffer[MAXLINE];
    int count;
```

- **Parametri:**
  - `int socket`: Il file descriptor per il socket attraverso cui avviene la comunicazione con il server.
  - `int id_studente`: L'ID dello studente, anche se in questa porzione di codice non viene usato, potrebbe essere utile per altre funzioni di gestione della prenotazione.
- **Variabili locali:**
  - `int scelta`: Usata per memorizzare l'opzione che l'utente seleziona.
  - `int nwrite`: Contiene il numero di byte scritti sul socket per verificare se la richiesta al server è andata a buon fine.
  - `char buffer[MAXLINE]`: Buffer utilizzato per memorizzare stringhe (anche se in questa parte non viene effettivamente usato).
  - `int count`: Memorizza il numero di esami disponibili ricevuto dalla risposta del server.

Richiede la lista degli esami disponibili

```
    // Richiede al server la lista degli esami disponibili
    printf("Scegliere l'opzione:\n");
    nwrite = FullWrite(socket, "MostraEsami_2", strlen("MostraEsami_2"));
```

- `printf("Scegliere l'opzione:\n")`: Stampa un messaggio per indicare all'utente che deve effettuare una scelta (anche se la scelta avviene più tardi).
- `nwrite = FullWrite(socket, "MostraEsami_2", strlen("MostraEsami_2"))`: Invio della richiesta `"MostraEsami_2"` al server. Questo comando presumibilmente chiede al server di inviare la lista degli esami disponibili.

- `FullWrite`: La funzione `FullWrite`, descritta in precedenza, si assicura che tutti i byte della stringa `"MostraEsami_2"` vengano inviati correttamente al server, anche se ci fossero interruzioni.
- `strlen("MostraEsami_2")`: Determina la lunghezza della stringa da inviare.

Gestione degli errori di scrittura

```
if (nwrite < 0) {
    printf("Errore in scrittura\n");
    return;
}
```

- Verifica se la scrittura sul socket è avvenuta correttamente.
- `if (nwrite < 0)`: Se `nwrite` è negativo, significa che c'è stato un errore nell'invio dei dati.
- `printf("Errore in scrittura\n")`: Stampa un messaggio di errore per indicare che la scrittura è fallita.
- `return`: Se c'è un errore, la funzione termina e non prosegue.

Legge la risposta dal server

```
char recvbuff[MAXLINE];
ssize_t nread = read(socket, recvbuff, sizeof(recvbuff) - 1);
```

- `char recvbuff[MAXLINE]`: Dichiarazione di un buffer chiamato `recvbuff` che verrà usato per memorizzare la risposta del server.
- `nread = read(socket, recvbuff, sizeof(recvbuff) - 1)`: La funzione `read` legge la risposta del server attraverso il socket e la memorizza in `recvbuff`. Viene letto un massimo di `sizeof(recvbuff) - 1` byte per lasciare spazio al carattere di terminazione `\0`.

Gestione degli errori di lettura

```
    if (nread < 0) {
        perror("Errore in lettura");
        return;
    }
```

- `if (nread < 0)`: Se `nread` è negativo, c'è stato un errore durante la lettura.
- `perror("Errore in lettura")`: Usa la funzione `perror` per stampare un messaggio di errore più dettagliato, fornendo anche informazioni specifiche sull'errore (usando il valore di `errno`).
- `return`: La funzione termina in caso di errore.

Termina la stringa ricevuta e analizza il numero di esami

```
    recvbuff[nread] = '\0'; // Termina la stringa
    sscanf(recvbuff, "Numero di esami: %d", &count); // Estrae il numero di esami disponibili
```

- `recvbuff[nread] = '\0'`: Dopo aver letto i dati dal server, si aggiunge un carattere di terminazione `\0` alla fine della stringa, per assicurarsi che `recvbuff` sia una stringa valida e correttamente terminata.
- `sscanf(recvbuff, "Numero di esami: %d", &count)`: Estrae il numero di esami disponibili dalla risposta del server usando `sscanf`. Si presume che il server invii una stringa con un formato del tipo `"Numero di esami: <numero>"`, e `sscanf` analizza questa stringa per ottenere il numero di esami, che viene memorizzato in `count`.

Mostra la risposta del server

```
    fputs(recvbuff, stdout); // Stampa la risposta del server
    printf("\n");
```

- `fputs(recvbuff, stdout)`: Stampa il contenuto di `recvbuff` (la risposta del server) sullo schermo.
- `printf("\n")`: Aggiunge una nuova linea per formattare meglio l'output.

Gestione della scelta dell'utente

```
    if (scanf("%d", &scelta) != 1) {
        printf("Input non valido.\n");
        while (getchar() != '\n'); // Pulisce il buffer di input
    }
```

- `scanf("%d", &scelta)`: Chiede all'utente di inserire un'opzione (il numero dell'esame da selezionare).
  - Se `scanf` non riesce a leggere un intero valido (ad esempio, se l'utente inserisce testo non numerico), restituisce un valore diverso da `1`.
- `if (scanf("%d", &scelta) != 1)`: Verifica se l'input dell'utente è valido. Se l'input non è un numero intero valido, la condizione è vera.
- `printf("Input non valido.\n")`: Stampa un messaggio di errore in caso di input non valido.
- `while (getchar() != '\n')`: Questo ciclo viene usato per svuotare il buffer di input nel caso in cui l'utente inserisca dati non validi. Continua a leggere caratteri fino a quando non incontra un carattere di nuova linea (`\n`), il che significa che tutti i caratteri non validi sono stati eliminati.

Verifica della scelta e gestione degli errori

```
    if (scelta == 0) {
        return; // Torna al menu principale
    }
    if (scelta > count) {
        system("clear");
        printf("Input non valido.\n");
        return;
    }
```

- `if (scelta == 0)`: Se l'utente inserisce `0`, la funzione termina e si presume che l'utente torni al menu principale o annulli l'operazione.
- `if (scelta > count)`: Se l'utente inserisce una scelta maggiore del numero di esami disponibili (`count`), significa che ha selezionato un'opzione non valida.
  - `system("clear")`: Pulisce lo schermo (sotto Unix/Linux, il comando `clear` è usato per cancellare il terminale).

- `printf("Input non valido.\n")`: Stampa un messaggio di errore per indicare che l'input non è valido.
- `return`: Termina la funzione.

Invia la richiesta di prenotazione

```
snprintf(buffer, sizeof(buffer), "%d:%d_2", id_studente, scelta - 1);
nwrite = FullWrite(socket, buffer, strlen(buffer));
```

- `snprintf(buffer, sizeof(buffer), "%d:%d_2", id_studente, scelta - 1)`:
  - **Cos'è**: Si crea una stringa formattata che rappresenta la richiesta di prenotazione dell'esame.
  - **Dettagli**:
    - `snprintf`: Funzione sicura che formatta una stringa e la memorizza nel buffer. Usa il buffer per evitare overflow, specificando la dimensione massima di `buffer` con `sizeof(buffer)`.
    - `"%d:%d_2"`: Questa è la stringa di formato. Viene utilizzata per formattare due numeri interi, separati da due punti (`:`), e aggiungere `_2` alla fine.
      - `id_studente`: L'ID dello studente, viene inserito al posto del primo `%d`.
      - `scelta - 1`: La scelta dell'utente, diminuita di 1 per convertire da input umano a indice basato su 0 (nel caso in cui l'esame scelto sia indicizzato in una lista a partire da 0). Questo valore viene inserito nel secondo `%d`.
    - Il risultato potrebbe essere una stringa come `"12345:2_2"`, dove `12345` è l'ID dello studente e `2` è l'indice dell'esame scelto.
- `nwrite = FullWrite(socket, buffer, strlen(buffer))`:
  - `FullWrite`: Viene utilizzata per inviare la stringa `buffer` attraverso il socket al server.
    - `socket`: Il file descriptor del socket.
    - `buffer`: La stringa formattata che rappresenta la richiesta di prenotazione.
    - `strlen(buffer)`: La lunghezza della stringa da inviare, calcolata con `strlen`.
  - **Funzione** `FullWrite`: Come spiegato in precedenza, `FullWrite` gestisce l'invio completo dei dati sul socket, riprovando in caso di interruzioni e assicurandosi che tutti i byte siano inviati.

Gestione degli errori di scrittura

```
if (nwrite < 0) {
    printf("Errore in scrittura\n");
    return;
}
```

- `if (nwrite < 0)`: Verifica se c'è stato un errore nell'invio della richiesta.
  - `nwrite < 0` indica che l'invio della stringa al server tramite il socket non è andato a buon fine.
- **Gestione dell'errore**:
  - `printf("Errore in scrittura\n")`: Viene mostrato un messaggio di errore all'utente.
  - `return`: La funzione termina immediatamente, interrompendo il processo di prenotazione.

Legge la conferma della prenotazione

```
nread = read(socket, recvbuff, sizeof(recvbuff) - 1);
```

- `nread = read(socket, recvbuff, sizeof(recvbuff) - 1)`:
  - Si tenta di leggere la risposta del server (la conferma della prenotazione) attraverso il socket.
  - `read`: La funzione `read` legge fino a `sizeof(recvbuff) - 1` byte dal socket e li memorizza in `recvbuff`. Il motivo per cui si legge `sizeof(recvbuff) - 1` è per riservare un byte per il carattere di terminazione `\0`.
  - `recvbuff`: Buffer di caratteri in cui viene memorizzata la risposta del server.
  - `nread`: Il numero di byte letti dal socket. Se è negativo, significa che c'è stato un errore durante la lettura.

Gestione degli errori di lettura

```
if (nread < 0) {
    perror("Errore in lettura");
    return;
}
```

- `if (nread < 0)`: Verifica se c'è stato un errore durante la lettura della risposta.

- `nread < 0` indica un errore di lettura sul socket, ad esempio una disconnessione o un problema di comunicazione con il server.

- **Gestione dell'errore**:
  - `perror("Errore in lettura")`: Stampa un messaggio di errore più dettagliato, includendo anche informazioni sul tipo specifico di errore (utilizzando `errno`).
  - `return`: La funzione termina immediatamente, interrompendo il processo.

Pulizia dello schermo

```
system("clear");
```

- `system("clear")`: Esegue il comando `clear` del sistema operativo per pulire lo schermo.
  - Questo è utile per rimuovere vecchie informazioni dal terminale e mostrare solo la conferma della prenotazione in maniera chiara.
  - Questo comando è generalmente disponibile su sistemi Unix/Linux, ma su Windows potrebbe essere necessario usare un comando diverso, come `system("cls")`.

Termina la stringa ricevuta e stampa la conferma del server

```
recvbuff[nread] = '\0'; // Termina la stringa
fputs(recvbuff, stdout); // Stampa la risposta del server
```

- `recvbuff[nread] = '\0'`:
  - Dopo la lettura, aggiunge un carattere di terminazione (`\0`) alla fine della stringa `recvbuff`. Questo garantisce che `recvbuff` sia una stringa valida e correttamente formattata.
  - È importante ricordare che i dati letti da `read` non includono automaticamente il carattere di terminazione, quindi deve essere aggiunto manualmente.
- `fputs(recvbuff, stdout)`:
  - `fputs`: Funzione che scrive la stringa `recvbuff` sullo schermo (stdout).
  - Mostra la risposta del server all'utente. Presumibilmente, il server invia una conferma della prenotazione o un messaggio di errore (ad esempio, "Prenotazione riuscita" o "Errore nella prenotazione").

Funzione `Esami_Disponibili`

```
void Esami_Disponibili(int socket) {
    int scelta;
    int nwrite;
```

- `int socket`: Questo parametro rappresenta il file descriptor del socket, attraverso il quale il client comunica con il server.
- `int scelta`: Variabile per memorizzare la scelta dell'utente.
- `int nwrite`: Variabile per tenere traccia del numero di byte inviati al server e verificare eventuali errori durante la scrittura sul socket.

Ciclo principale

```
while (1) {
```

- `while (1)`: Il ciclo infinito che mantiene la funzione attiva finché l'utente non decide di tornare indietro o terminare. Ogni volta che l'utente seleziona un'opzione, viene inviata una richiesta al server per recuperare informazioni sugli esami disponibili.

Menu delle opzioni degli esami

```
    // Mostra il menu delle opzioni degli esami
    printf("Date Esami:\n");
    printf("Scegliere l'esame:\n");
    printf("1) Reti di Calcolatori\n");
    printf("2) Algoritmi e Strutture Dati\n");
    printf("3) Programmazione 1\n");
    printf("4) Programmazione 2\n");
    printf("5) Programmazione 3\n");
    printf("6) Tecnologie Web\n");
    printf("0) Torna Indietro\n");
```

- `printf`: Stampa una serie di opzioni sul terminale, ognuna delle quali rappresenta un esame che l'utente può selezionare.

- Opzioni da 1 a 6: Sono esami specifici (ad esempio, "Reti di Calcolatori", "Programmazione 1", ecc.).
- Opzione 0: Permette all'utente di tornare indietro o uscire dal menu.

Input dell'utente e gestione dell'input non valido

```
if (scanf("%d", &scelta) != 1) {
    printf("Input non valido.\n");
    while (getchar() != '\n'); // Pulisce il buffer di input
    continue;
}
```

- `scanf("%d", &scelta)`: Chiede all'utente di inserire un numero intero corrispondente a una delle opzioni del menu.
  - Se `scanf` non riesce a leggere un numero intero valido (ad esempio, se l'utente inserisce testo invece di numeri), restituisce un valore diverso da `1`.
- **Controllo dell'input**:
  - `if (scanf(...) != 1)`: Verifica se l'input dell'utente è valido. Se non è un numero, si entra nell' `if`.
  - **Gestione dell'input non valido**:
    - `printf("Input non valido.\n")`: Viene stampato un messaggio di errore.
    - `while (getchar() != '\n')`: Si svuota il buffer di input, eliminando eventuali caratteri errati rimasti.
    - `continue`: Riprende il ciclo dall'inizio, chiedendo nuovamente all'utente di inserire una scelta.

Invia la richiesta di informazioni sugli esami

```
switch (scelta) {
    case 1:
        system("clear");
        nwrite = FullWrite(socket, "Reti_1", strlen("Reti_1"));
        break;
    case 2:
        system("clear");
        nwrite = FullWrite(socket, "Algoritmi_1", strlen("Algoritmi_1"));
        break;
```

```
        case 3:
            system("clear");
            nwrite = FullWrite(socket, "Prog1_1", strlen("Prog1_1"));
            break;
        case 4:
            system("clear");
            nwrite = FullWrite(socket, "Prog2_1", strlen("Prog2_1"));
            break;
        case 5:
            system("clear");
            nwrite = FullWrite(socket, "Prog3_1", strlen("Prog3_1"));
            break;
        case 6:
            system("clear");
            nwrite = FullWrite(socket, "Web_1", strlen("Web_1"));
            break;
        case 0:
            system("clear");
            return; // Torna al menu principale
        default:
            system("clear");
            printf("Opzione non disponibile");
            return;
    }
```

- `switch (scelta)`: Questo `switch` verifica quale opzione è stata selezionata dall'utente.
  - **Case da 1 a 6**: Ciascun `case` corrisponde a uno specifico esame. Quando l'utente seleziona un'opzione:
    - `system("clear")`: Pulisce lo schermo del terminale (su sistemi Unix/Linux) per migliorare la visualizzazione.
    - `nwrite = FullWrite(...)`: Invia una stringa al server per richiedere informazioni sull'esame selezionato. Ogni stringa ha un formato univoco come `"Reti_1"`, `"Algoritmi_1"`, ecc. Questo comando viene inviato al server tramite il socket.
      - `FullWrite`: Funzione che garantisce l'invio completo della stringa attraverso il socket (già descritta in precedenza).
  - **Case 0**: Se l'utente seleziona l'opzione `0`, la funzione chiama nuovamente `system("clear")` e termina, riportando l'utente al menu principale.
  - **Default**: Se l'utente seleziona un'opzione non valida (ad esempio un numero che non è nel range 0-6):
    - `system("clear")`: Pulisce lo schermo.
    - `printf("Opzione non disponibile")`: Mostra un messaggio di errore.
    - `return`: Termina la funzione, riportando l'utente al menu principale.

Gestione degli errori di scrittura

```
    if (nwrite < 0) {
        printf("Errore in scrittura\n");
        return;
    }
```

- `if (nwrite < 0)`: Verifica se l'invio della richiesta al server è avvenuto con successo.
  - Se `nwrite` è negativo, significa che c'è stato un errore durante l'invio dei dati.
  - **Gestione degli errori**:
    - `printf("Errore in scrittura\n")`: Stampa un messaggio di errore.
    - `return`: Termina la funzione in caso di errore, senza procedere oltre.

Legge e stampa la lista degli esami disponibili

```
    char recvbuff[MAXLINE];
    ssize_t nread = read(socket, recvbuff, sizeof(recvbuff) - 1);
    if (nread < 0) {
        perror("Errore in lettura");
        return;
    }
    recvbuff[nread] = '\0'; // Termina la stringa
    fputs(recvbuff, stdout); // Stampa la risposta del server
    printf("\n");
    return;
```

- `char recvbuff[MAXLINE]`: Viene dichiarato un buffer per memorizzare la risposta del server.
- `nread = read(socket, recvbuff, sizeof(recvbuff) - 1)`:
  - Legge la risposta del server attraverso il socket e la memorizza in `recvbuff`.
  - Il motivo per cui si legge `sizeof(recvbuff) - 1` byte è per riservare spazio per il carattere di terminazione `\0`.
- **Gestione degli errori di lettura**:
  - `if (nread < 0)`: Se la lettura non riesce, viene stampato un messaggio di errore con `perror`, e la funzione termina con `return`.
- **Aggiunta del carattere di terminazione**:

    - `recvbuff[nread] = '\0'` : Dopo aver letto i dati, si aggiunge un carattere di terminazione `\0` alla fine della stringa per assicurarsi che sia una stringa valida.
- **Stampa della risposta**:
    - `fputs(recvbuff, stdout)` : Stampa la risposta del server (presumibilmente una lista di date o informazioni sugli esami) sullo schermo.
    - `printf("\n")` : Aggiunge una nuova riga per migliorare la leggibilità.
- `return` : Termina la funzione dopo aver stampato la risposta del server.

Dichiarazione della funzione e variabili

```
void Student_Function(FILE *filein, int socket, int id_studente) {
    char sendbuff[MAXLINE + 1], recvbuff[MAXLINE + 1];
    int scelta;
    fd_set fset;
    int maxfd;
    system("clear"); // Pulisce lo schermo
```

- **Parametri**:
    - `FILE *filein` : Questo parametro rappresenta il file di input da cui leggere i dati dell'utente, di solito è `stdin` per l'input standard.
    - `int socket` : Il file descriptor del socket attraverso il quale il client comunica con il server.
    - `int id_studente` : L'ID dello studente. Potrebbe essere usato per identificare lo studente in operazioni successive, come richieste di prenotazione.
- **Variabili locali**:
    - `sendbuff` **e** `recvbuff` : Buffer usati per memorizzare i dati da inviare al server e quelli ricevuti dal server. La dimensione è basata su `MAXLINE`, e si aggiunge 1 per il carattere di terminazione della stringa `\0`.
    - `scelta` : Variabile che memorizza la scelta dell'utente nel menu.
    - `fd_set fset` : Set di file descriptor usato con la funzione `select` per monitorare più file descriptor (in questo caso il socket e l'input dell'utente).
    - `maxfd` : Variabile che tiene traccia del file descriptor più alto tra quelli monitorati.
- `system("clear")` : Pulisce lo schermo del terminale per fornire un'interfaccia più ordinata.

Ciclo principale

```
    while (1) {
```

- `while (1)`: Un ciclo infinito che continua fino a quando l'utente non seleziona l'opzione di uscita (`0`), o il server chiude la connessione.

Mostra il menu delle opzioni

```
        // Mostra il menu delle opzioni
        printf("Scegliere l'opzione:\n");
        printf("1) Esami disponibili per corso\n");
        printf("2) Richiedi prenotazione esame\n");
        printf("0) Uscire\n");
```

- `printf`: Mostra il menu delle opzioni, permettendo all'utente di:
  - **1**: Visualizzare gli esami disponibili.
  - **2**: Richiedere la prenotazione di un esame.
  - **0**: Uscire dal programma.

Inizializzazione del set di file descriptor

```
        // Inizializzazione del set di file descriptor per select
        FD_ZERO(&fset);
        FD_SET(socket, &fset);          // Aggiungi il socket al set
        FD_SET(fileno(filein), &fset);  // Aggiungi l'input standard al set
        maxfd = max(fileno(filein), socket) + 1; // Trova il massimo file
descriptor
```

- `FD_ZERO(&fset)`: Inizializza il set di file descriptor vuoto. Questo set sarà monitorato da `select`.
- `FD_SET(socket, &fset)`: Aggiunge il file descriptor del socket al set. Significa che il programma monitorerà il socket per vedere se ci sono dati in arrivo.
- `FD_SET(fileno(filein), &fset)`: Aggiunge il file descriptor associato all'input (di solito `stdin`, ottenuto con `fileno(filein)`) al set. Significa che il programma monitorerà anche l'input dell'utente.
- `maxfd = max(fileno(filein), socket) + 1`: Trova il file descriptor più grande tra il socket e l'input standard, e aggiunge 1. Questo è necessario perché `select` richiede di sapere

qual è il massimo file descriptor che deve monitorare.

Attesa di input o dati dal server

```
        // Attende che uno dei file descriptor nel set diventi pronto
        if (select(maxfd, &fset, NULL, NULL, NULL) < 0) {
            perror("Select error");
            return;
        }
```

- `select(maxfd, &fset, NULL, NULL, NULL)`:
  - La funzione `select` attende che uno dei file descriptor diventi "pronto" per la lettura (input dell'utente o dati in arrivo dal server).
  - **Parametri**:
    - `maxfd`: Il massimo file descriptor da monitorare.
    - `&fset`: Il set di file descriptor che `select` deve monitorare per l'input.
    - I valori `NULL` per gli altri parametri indicano che non si monitorano file descriptor per la scrittura o errori, e che non c'è un timeout.
- **Gestione degli errori**:
  - `if (select(...) < 0)`: Se `select` restituisce un valore negativo, significa che c'è stato un errore.
  - `perror("Select error")`: Stampa un messaggio di errore.
  - `return`: Esce dalla funzione in caso di errore.

Controllo se l'utente ha inserito un input

```
        // Controlla se ci sono dati disponibili sull'input standard
        if (FD_ISSET(fileno(filein), &fset)) {
            if (scanf("%d", &scelta) != 1) {
                printf("Input non valido.\n");
                while (getchar() != '\n'); // Pulisce il buffer di input
                continue;
            }
```

- `FD_ISSET(fileno(filein), &fset)`: Verifica se l'input dell'utente è pronto (cioè se l'utente ha inserito qualcosa).

    - Se l'input è pronto, il programma entra in questo blocco.
- `scanf("%d", &scelta)`: Legge la scelta dell'utente e la memorizza nella variabile `scelta`.
    - **Gestione degli input non validi**:
        - Se l'input non è un numero valido, `scanf` restituisce un valore diverso da 1, e si stampa il messaggio `"Input non valido.\n"`.
        - `while (getchar() != '\n')`: Svuota il buffer di input nel caso in cui l'utente abbia inserito dati non validi.

Gestione delle scelte dell'utente

```
            // Gestione delle opzioni dell'utente
            switch (scelta) {
                case 1:
                    system("clear");
                    Esami_Disponibili(socket); // Mostra gli esami disponibili
                    break;
                case 2:
                    system("clear");
                    Richiesta_Prenotazione(socket, id_studente); // Richiede la
prenotazione dell'esame
                    break;
                case 0:
                    return; // Esce dal loop e termina il programma
                default:
                    printf("Opzione non valida.\n");
                    break;
            }
```

- `switch (scelta)`: Gestisce l'opzione scelta dall'utente.
    - **Case 1**: Se l'utente seleziona "1", chiama la funzione `Esami_Disponibili(socket)`, che mostra gli esami disponibili dal server.
    - **Case 2**: Se l'utente seleziona "2", chiama la funzione `Richiesta_Prenotazione(socket, id_studente)`, che invia una richiesta di prenotazione esame.
    - **Case 0**: Se l'utente seleziona "0", esce dal ciclo e termina la funzione, con il programma che probabilmente torna al menu principale o chiude.
    - **Default**: Se l'utente inserisce una scelta non valida, stampa `"Opzione non valida.\n"`.

Controllo se ci sono dati in arrivo dal server

```
        if (FD_ISSET(socket, &fset)) {
            // Verifica se il socket è stato chiuso
            ssize_t nread = read(socket, recvbuff, sizeof(recvbuff) - 1);
            if (nread == 0) {
                // Il server ha chiuso la connessione
                system("clear");
                printf("Il server ha chiuso la connessione.\n");
                return;
            }
            if (nread < 0) {
                perror("Errore in lettura dal socket");
                return;
            }
        }
```

- `FD_ISSET(socket, &fset)` : Verifica se ci sono dati in arrivo dal server (attraverso il socket).
  - Se il socket è pronto, il programma entra in questo blocco.
- `ssize_t nread = read(socket, recvbuff, sizeof(recvbuff) - 1)` : Legge i dati dal server e li memorizza nel buffer `recvbuff` .
  - **Gestione della chiusura della connessione**:
    - `if (nread == 0)` : Se `nread` è pari a 0, significa che il server ha chiuso la connessione.
      - Il programma pulisce lo schermo e stampa `"Il server ha chiuso la connessione.\n"` , poi termina la funzione.
  - **Gestione degli errori di lettura**:
    - `if (nread < 0)` : Se c'è un errore durante la lettura dei dati, stampa un messaggio di errore e termina la funzione.

Main

Dichiarazione delle variabili

```
int sock;
struct sockaddr_in serv_add;
```

- `int sock` : Questo rappresenta il file descriptor del socket, che verrà usato per comunicare con il server.

- `struct sockaddr_in serv_add`: Una struttura che contiene le informazioni relative all'indirizzo del server, come il tipo di protocollo (IPv4), il numero di porta, e l'indirizzo IP.

Inizializzazione del generatore di numeri casuali

```
// Inizializzazione del generatore di numeri casuali
srand(time(NULL));
int id_studente = rand(); // Assegna un ID casuale allo studente
```

- `srand(time(NULL))`: Inizializza il generatore di numeri casuali con il valore corrente del tempo. Questo garantisce che ogni esecuzione del programma generi un ID studente diverso.
- `int id_studente = rand();`: Genera un numero casuale, che viene usato come ID dello studente. Questo ID è probabilmente utilizzato per identificare l'utente nel sistema.

Verifica degli argomenti

```
// Verifica che l'argomento dell'indirizzo IP sia fornito
if (argc != 2) {
    fprintf(stderr, "Usage: %s <server-ip-address>\n", argv[0]);
    return 1;
}
```

- `argc`: Rappresenta il numero di argomenti passati al programma dalla riga di comando.
- `argv`: Un array di stringhe che contiene gli argomenti. Il primo (`argv[0]`) è il nome del programma, il secondo (`argv[1]`) dovrebbe essere l'indirizzo IP del server.
- **Verifica**: Il programma controlla che venga passato esattamente un argomento (l'indirizzo IP del server). Se questo argomento manca, stampa un messaggio di errore e termina con `return 1`.

Apertura del socket

```
// Apertura del socket
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
```

```
    perror("Socket creation error");
    return 1;
}
```

- `socket(AF_INET, SOCK_STREAM, 0)` : Crea un nuovo socket.
  - `AF_INET` : Indica che si utilizza il protocollo IPv4.
  - `SOCK_STREAM` : Indica che si sta creando un socket di tipo TCP, basato su una connessione affidabile.
  - `0` : Specifica il protocollo predefinito (TCP, nel caso di `SOCK_STREAM`).
- **Verifica**: Se la creazione del socket fallisce (`sock < 0`), viene stampato un messaggio di errore con `perror` e il programma termina.

Impostazione dell'indirizzo del server

```
// Imposta i campi della struttura sockaddr_in per la connessione al server
serv_add.sin_family = AF_INET;
serv_add.sin_port = htons(54321); // Porta del server
```

- `serv_add.sin_family = AF_INET` : Indica che il socket utilizza l'indirizzo IPv4.
- `serv_add.sin_port = htons(54321)` : Imposta la porta del server a `54321` . La funzione `htons` (host to network short) converte il numero di porta dall'ordine dei byte dell'host all'ordine dei byte di rete, garantendo la corretta interpretazione indipendentemente dall'architettura della macchina.

Conversione dell'indirizzo IP

```
// Converte l'indirizzo IP da stringa a formato binario
if (inet_pton(AF_INET, argv[1], &serv_add.sin_addr) <= 0) {
    perror("Address creation error");
    close(sock);
    return 1;
}
```

- `inet_pton(AF_INET, argv[1], &serv_add.sin_addr)` : Converte l'indirizzo IP in formato stringa (passato come argomento) nel formato binario adatto per `sockaddr_in` e lo assegna a `serv_add.sin_addr` .

    - `argv[1]`: Contiene l'indirizzo IP del server (es: "192.168.1.1").
    - `&serv_add.sin_addr`: Campo dell'indirizzo IP in formato binario.
- **Verifica**: Se la conversione fallisce (ad esempio, perché l'indirizzo IP non è valido), viene stampato un messaggio di errore con `perror`, il socket viene chiuso con `close(sock)`, e il programma termina.

Connessione al server

```
// Connessione al server
if (connect(sock, (struct sockaddr *)&serv_add, sizeof(serv_add)) < 0) {
    perror("Connection error");
    close(sock);
    return 1;
}
```

- `connect(sock, (struct sockaddr *)&serv_add, sizeof(serv_add))`: Tenta di stabilire una connessione TCP al server usando il socket appena creato.
    - `sock`: Il file descriptor del socket.
    - `(struct sockaddr *)&serv_add`: L'indirizzo del server a cui connettersi.
    - `sizeof(serv_add)`: La dimensione della struttura `sockaddr_in`.
- **Verifica**: Se la connessione fallisce, viene stampato un messaggio di errore con `perror`, il socket viene chiuso, e il programma termina.

Funzioni dello studente

```
// Funzioni dello studente per interagire con il server
Student_Function(stdin, sock, id_studente);
```

- `Student_Function(stdin, sock, id_studente)`: Chiama una funzione che gestisce l'interazione tra lo studente e il server. Viene passato l'input standard (`stdin`), il socket per la comunicazione e l'ID dello studente.
    - Questa funzione gestirà probabilmente il ciclo principale dell'interazione (come inviare richieste, leggere risposte dal server, ecc.).

Chiusura del socket

```
// Chiusura del socket alla fine
close(sock);
return 0;
```

- `close(sock)` : Chiude il socket una volta terminata la comunicazione con il server.
- `return 0` : Il programma termina con successo restituendo `0` .