ChatterBox

Luca Canessa (516639)

Version 1.0.0
Tue Apr 3 2018

Table of Contents

Bug List

Global clear_queue (queue_t *q)
Erases all node but not the last

Data Structure Index

Data Structures

Here are the data structures with brief descriptions:

File Index

File List

Here is a list of all documented files with brief descriptions:

Data Structure Documentation

arg_rq_hand Struct Reference

Data Fields

- hashtable_t * **users**
- struct pollfd * **con**
- threadpool_t * **pool**
- int **i_fd**
- int **fd**

The documentation for this struct was generated from the following files:

- bk.chatty.c
- chatty.c

data Struct Reference

body del messaggio

Detailed Description

body del messaggio

The documentation for this struct was generated from the following file:

- message.h

elem Struct Reference

Data Fields

- user_t * **user**
- queue_t * **collision**

The documentation for this struct was generated from the following file:

- hashtable.h

---

gr Struct Reference

Data Fields

- group_chat_t * **group**
- queue_t * **collision**

The documentation for this struct was generated from the following file:

- hashtable.h

---

grp_msg Struct Reference

Data Fields

- char **chat_title** [MAX_NAME_LENGTH]
- queue_t * **participants**
- queue_t * **messages**
- char **creator** [MAX_NAME_LENGTH]

The documentation for this struct was generated from the following file:

- message.h

---

header Struct Reference

header del messaggio

Detailed Description

header del messaggio

header della parte dati

The documentation for this struct was generated from the following file:

- message.h

---

ht Struct Reference

Data Fields

- pthread_mutex_t **ht_lock**
- ht_elem_t * **users**
- ht_G_t * **groups**
- queue_t * **active_user**
- int **max_u**
- int **reg_users**

The documentation for this struct was generated from the following file:

- hashtable.h

---

message_data_hdr_t Struct Reference

Data Fields

- char **receiver** [MAX_NAME_LENGTH+1]
- unsigned int **len**

The documentation for this struct was generated from the following file:

- message.h

---

message_data_t Struct Reference

Data Fields

- message_data_hdr_t **hdr**
- char * **buf**

The documentation for this struct was generated from the following file:

- message.h

---

message_hdr_t Struct Reference

Data Fields

- op_t **op**
- char **sender** [MAX_NAME_LENGTH+1]

The documentation for this struct was generated from the following file:

- message.h

---

message_t Struct Reference

Data Fields

- message_hdr_t **hdr**
- message_data_t **data**

The documentation for this struct was generated from the following file:

- message.h

---

messaggio Struct Reference

tipo del messaggio

Detailed Description

tipo del messaggio

The documentation for this struct was generated from the following file:

- message.h

---

node Struct Reference

Data Fields

- void * **ptr**
- struct node * **next**
- struct node * **prev**

The documentation for this struct was generated from the following file:

- queue.h

---

of_arg Struct Reference

Data Fields

- char * **path**
- FILE * **f**

The documentation for this struct was generated from the following files:

- bk.chatty.c
- chatty.c

---

operation_t Struct Reference

Data Fields

- char * sname
  nickname del sender
- char * rname
  nickname o groupname del receiver
- op_t op
  tipo di operazione (se OP_END e' una operazione interna)
- char * msg
  messaggio testuale o nome del file
- long size
  lunghezza del messaggio
- long n
  usato per -R -r

Field Documentation

char* operation_t::msg

messaggio testuale o nome del file

long operation_t::n

usato per -R -r

op_t operation_t::op

tipo di operazione (se OP_END e' una operazione interna)

char* operation_t::rname

nickname o groupname del receiver

long operation_t::size

lunghezza del messaggio

char* operation_t::sname

nickname del sender

The documentation for this struct was generated from the following file:

- client.c

pool Struct Reference

Data Fields

- pthread_t * **thread**
- queue_t * **task**
- pthread_mutex_t **lock_t**
- pthread_cond_t **cond_t**
- int **count**
- int **shutdown**

The documentation for this struct was generated from the following file:

- pool.h

queue Struct Reference

Data Fields

- node_t * **head**
- node_t * **tail**
- unsigned long **queue_len**
- pthread_mutex_t **queue_lock**

The documentation for this struct was generated from the following file:

- queue.h

s_conf Struct Reference

Data Fields

- char **StatFileName** [256]
- char **DirName** [256]
- char **UnixPath** [256]
- int **MaxFileSize**
- int **MaxConnections**
- int **ThreadsInPool**
- int **MaxMsgSize**
- int **MaxHistMsgs**

The documentation for this struct was generated from the following file:

- config.h

statistics Struct Reference

Data Fields

- unsigned long **nusers**
- unsigned long **nonline**
- unsigned long **ndelivered**
- unsigned long **nnotdelivered**
- unsigned long **nfiledelivered**
- unsigned long **nfilenotdelivered**
- unsigned long **nerrors**
- pthread_mutex_t **statLock**

The documentation for this struct was generated from the following file:

- stats.h

---

struct Struct Reference

Structure to save the socket info, and info to manage the requests.

Detailed Description

Structure to save the socket info, and info to manage the requests.

Define the data structure to implement queue.

Defines the data structure for queue's nodes.

Defines a new structure type 'threadpool_t' to implements the thread pool.

Defines a new type of 'thread_task_t' structure to implement the work to be run.

Defines a new structure of type 'group_chat_t' to implement group chats.

Describes the entire system where save users and groups.

Describes the group hash table element.

Describes the user hash table element.

Describes an user with the data structures.

Defines the server configuration variables.

Structure used to manage file to send.

The documentation for this struct was generated from the following file:

- chatty.c

---

task Struct Reference

Data Fields

- void(* **function**)(void *)
- void * **args**

The documentation for this struct was generated from the following file:

- pool.h

---

us Struct Reference

Data Fields

- char **nickname** [MAX_NAME_LENGTH]
- int **fd_online**
- unsigned int **alrdy_read**
- queue_t * **chats**
- queue_t * **mygroup**
- pthread_mutex_t **us_lock**

The documentation for this struct was generated from the following file:

- hashtable.h

File Documentation

chatty.c File Reference

File principale del server chatterbox.

Data Structures

- struct arg_rq_hand
- struct of_arg

Macros

- #define **_POSIX_C_SOURCE** 200809L

Typedefs

- typedef struct arg_rq_hand **rq_arg**

Functions

- static void usage (const char *progname)
  usage Function to print the correct command that runs the server
- void print_statistic (hashtable_t *usr, char *FileName)
  print_statistic
- void configServer (struct s_conf *config)
  configServer
- void copyMSG (message_t *msg_dest, message_t *msg_src)
  copyMSG
- void cpyUsr (queue_t *usrs, char *buf)
  cpyUsr
- void sendUlist (queue_t *u, message_t *msg, user_t *me, char *buff)
  sendUlist
- int sendtoall (message_t *new, hashtable_t *users)

sendtoall

- void clear_chat (queue_t *chat)
  clear_chat
- char * rename_f (char *old)
  rename_f
- void requests_handler (void *args)
  request_handler
- void sig_handler (int sig)
  sig_handler
- int **main** (int argc, char *argv[])

Variables

- pthread_mutex_t main_l = PTHREAD_MUTEX_INITIALIZER
  mutex to block some data managed into this file
- static volatile int keepRUN = 1
  global variable to know when the signal is sent to shutdown the server
- static volatile int printLOG = 0
  global variable to know when the signal is sent to print the log
- struct statistics **chattyStats** = { 0,0,0,0,0,0,0, PTHREAD_MUTEX_INITIALIZER }

Detailed Description

File principale del server chatterbox.

$chatty_c$

LICENSE

Author:

Luca Canessa (516639) *

-

Copyright:

* Declares that all contents of this file are author's original operas *

-

DESCRIPTION

Looking overview

This file contains functions to manage requests from clients. The main function is "main ()" where sockets are set, to listen to requests from clients and the socket listener that accepts a connection. When the listener accepts a request, set up a socket to handle requests from a single client. Using a loop and the "poll ()" function, it can parse each socket to determine if a client has sent a request. For each request sent, "main ()" sets up a structure with information about the socket and inserts the request in the threaded task queue with this information. Each request is managed by the "request_handler ()" function. This function analyzes the OP element in the message and with the "switch" command can handle the specific request. The "main ()" function also handles signals using "sig_handler ()", which handles some sent signals.

usage() : Function used to print the command, to run the server, in the standard error. Requires the pointer to program path. Returns nothing

print_statistic() : Function used to print the server statistics into the log file after update some statistic. Requires the pointer to the hash table where saved the users and pointer to log file path. Returns nothing

configServer() : Function used to set the global varabiable that sets up the server. Requires the pointer to the structure where values are saved to configure the server. Returns nothing.

cpyMSG() : Function used to copy each element of the message from one to another. Requires the pointer to the destination message and the pointer to the source message. Returns nothing.

cpyUsr() : Function used to copy the active users list into to a buffer. Requires the pointer to active users queue and pointer to a buffer in which to save the list. Returns nothing

sendUlist() : Function used to set up the message to send with the active users. Requires the pointer to active users queue, pointer to message to send, pointer to user that receive the message and pointer to a buffer where save the users. Returns nothing.

sendtoall() : Function used to send a message to each registered user. If the user is active, the message is sent directly, otherwise it is entered in the history and the user can read it whenever he wishes. Requires the pointer to the message to be sent and the pointer to the hash table where the users were saved. Returns an integer: if equal to 0 then there were no errors, otherwise some users may not have received the message.

clear_chat() : Function used to erase each message in history chat Requires the pointer to history chat. Return nothing.

rename_f() : Function used to rename the file path that will be received. Check if other files have the same name, in this case it adds a number at the end that corresponds to the n-th file that will be saved. Requires the pointer to the original file path. Returns the pointer to the new path.

request_handler() : Function used to manage the requests sent from the clients. It reads the message sent and select the operation to do according to the "OP" message item. The "OP" codes are setted in the file "ops.h".

REGISTER_OP: registering an user CONNECT_OP: connecting an user POSTTXT_OP: receiving a message txt and resending it POSTTXTALL_OP: receiving and sending it to every user a txt message POSTFILE_OP: receiving a file GETFILE_OP: sending a file GETPREVMSGS_OP: sending the history chat USRLIST_OP: sending a active user list UNREGISTER_OP: deregistering an user DISCONNECT_OP: deactiveting an user CREATEGROUP_OP: creating a group ADDGROUP_OP: adding an user to a group DELGROUP_OP: removing an user from a group or delete the group GETPREVGROUPMSGS_OP: sending the group history chat

Requires the pointer to a structure where socket information is saved. Returns nothing.

sig_handler() : Function to manage the signals: SIGINT SIGTERM SIGQUIT SIGPIPE SIGUSR1 Requires signal code. Returns nothing.

Function Documentation

void clear_chat (queue_t * *chat*)

clear_chat

Function to clear the messages chat from user history queue

Parameters:

chat	Pointer to the user's message queue

void configServer (struct s_conf * *config*)

configServer

Function to update the server configuration variables

Parameters:

config	Pointer to the structure in which the variables are allocated

void copyMSG (message_t * *msg_dest*, message_t * *msg_src*)

copyMSG

Function used to copy the elements of a message from one to another

Parameters:

msg_dest	Pointer to destination message
msg_src	Pointer to source message

void cpyUsr (queue_t * *usrs*, char * *buf*)

cpyUsr

Function used to copy the active users from the queue to a buffer

Parameters:

usrs	Pointer to the active users' queue

void print_statistic (hashtable_t * *usr*, char * *FileName*)

print_statistic

Function to update structure items of statistic log and save this log

Parameters:

usr	Pointer to the hash table where users are saved
FileName	Pointer to the log path

char* rename_f (char * *old*)

rename_f

Function to change the path of the file to be received. If the file already exists, it also updates the name.

Parameters:

old	Pointer to the path of the original file received

Returns:

new Pointer to the new file path

void requests_handler (void * *args*)

request_handler

Function to manage all clients requests

Parameters:

args	Pointer to the data structure where the information of the request to be managed and the information of the socket used are saved

< REGISTER AN USER

< CONNECT AN USER

< SEND USER LIST

< DISCONNECT AN USER

< UNREGISTER AN USER

< CREATE A GROUP

< ADDITION OF A USER TO A GROUP

< DELETE A GROUP OR REMOVE A USER

< SEND MESSAGE RECEIVED TO A RECEIVER USER OR GROUP

< SEND A MESSAGE RECEIVED TO ALL USERS

< SEND FILE RECEIVED TO A RECEIVER USER OR GROUP

< SEND USER CHAT HISTORY

< SEND GROUP CHAT HISTORY

< SEND FILE TO THE RECEIVER USER

int sendtoall (message_t * *new*, hashtable_t * *users*)

sendtoall

Function to send a message to all registered users

Parameters:

new	Pointer to the message to be sent
users	Pointer to the hash table where users were saved

Returns:

err Integer value: if equal to 0 then no errors have occurred otherwise if less than 0 there may have been errors

void sendUlist (queue_t * *u*, message_t * *msg*, user_t * *me*, char * *buff*)

sendUlist

Function used to set the message to send to user with active user list

Parameters:

u	Pointer to users queue
msg	Pointer to the message to be set
me	Pointer to the user who must receive the message
buff	Pointer to the buffer to insert users

void sig_handler (int *sig*)

sig_handler

Function to manage the received signals

Parameters:

sig	Signal code

static void usage (const char * *progname*) [static]

usage Function to print the correct command that runs the server

Parameters:

progname	Pointer to server path

Variable Documentation

volatile int keepRUN = 1 [static]

global variable to know when the signal is sent to shutdown the server

pthread_mutex_t main_l = PTHREAD_MUTEX_INITIALIZER

mutex to block some data managed into this file

volatile int printLOG = 0 [static]

global variable to know when the signal is sent to print the log

client.c File Reference

Semplice client di test.

Data Structures

- struct operation_t

Macros

- #define **_POSIX_C_SOURCE** 200809L

Functions

- static void **use** (const char *filename)
- static int **readMessage** (int connfd, message_hdr_t *hdr)
- static int **downloadFile** (int connfd, char *filename, char *sender)
- static int **execute_requestreply** (int connfd, operation_t *o)
- static int **execute_receive** (int connfd, operation_t *o)
- int **main** (int argc, char *argv[])

Variables

- static message_t * **MSGS** = NULL
- static const int **msgbatch** = 100
- static size_t **msgcur** =0
- static size_t **msglen** =0

Detailed Description

Semplice client di test.

config.h File Reference

File contenente alcune define con valori massimi utilizzabili.

Data Structures

- struct s_conf

Macros

- #define **MAX_NAME_LENGTH** 32
- #define COMMENT '#'
  define the symbol that represents the beginning of the comments
- #define TOKEN "="
  defines the symbol that divide the config variable from its value
- #define NEW_LINE '\n'
  defines the symbol that represent a new line in the file

Typedefs

- typedef int **make_iso_compilers_happy**

Functions

- void pars (char *name, struct s_conf *s)
  pars

Detailed Description

File contenente alcune define con valori massimi utilizzabili.

Macro Definition Documentation

#define COMMENT '#'

define the symbol that represents the beginning of the comments

#define NEW_LINE '\n'

defines the symbol that represent a new line in the file

#define TOKEN "="

defines the symbol that divide the config variable from its value

Function Documentation

void pars (char * *name*, struct s_conf * *s*)

pars

Prende il file di configurazione e lo legge. Per ogni opzione conosciuta, selezionare il suo valore e lo salva nella variabile del server appropriata.

Parameters:

name	Pointer to path to the config file
s	Pointer to 's_conf' struct (see 's_conf' description for details)

Opens the file passed in read-only mode. Read each line and check that it is not a comment or a blank line (new line '

'). Then it divides the configuration option from its value. For each known option, save the value in the appropriate server configuration variable.

Parameters:

name	Pointer to configuration file path
s	Pointer to 's_conf' structure

connections.c File Reference

$connections_c$

Functions

- int openConnection (char *path, unsigned int ntimes, unsigned int secs)
  openConnection
- int readHeader (long connfd, message_hdr_t *hdr)
  readHeader
- int sendHeader (long connfd, message_hdr_t *hdr)
  sendHeader
- int sendRequest (long fd, message_t *msg)
  sendRequest
- int readData (long fd, message_data_t *data)
  readData
- int sendData (long fd, message_data_t *msg)
  sendData
- int readMsg (long fd, message_t *msg)
  readMsg

Variables

- int **_FILESIZE**
- int _MSGSIZE
  _FILESIZE is the max size of a file could be sent - _MSGSIZE is the max size of a message could be sent
- char SOCKNAME [UNIX_PATH_MAX]

path where is saved the socket

- char Dir [256]
  path where save the files received

Detailed Description

$connections_c$

LICENSE

Author:

Luca Canessa (516639) *

-

Copyright:

 * Declares that all contents of this file are author's original operas *

-

DESCRIPTION

Looking overview

In this file are present functions to manage connections from and to clients. Every communication between server and clients is handled using messages. Each request from clients is presents in the message header and into the message body, there are the arguments to manage the request. Each received datum is in char representation so each function that manages this, it must cast in datum type. Instead, if the datum shall be sent, the function that manges it, must save the datum in char into a buffer.

openConnection() : Its task is to open a file descriptor for a socket. Requires a pointer to the path where save the socket that will be open, the number of time to retry to connect if there are some problems and the seconds that must pass between the retrying. Returns the idetifier of file destriptor.

readHeader() : Receives a buffer where are present the message header items. Requires the identifier of file descriptor and pointer to a message header where save the data received. Returns the number of bytes read or -1 if exits with some errors.

sendHeader() : Sends a buffer where are present the message header items. Requires the identifier of file descriptor and the pointer to a message header where saved the data to be sent. Returns the number of bytes sent or -1 if exits with some errors.

sendRequest() : Sends a buffer where are present the message items. Requires the identifier of file descriptor and the pointer to a message where saved the data that will be managed by server. Returns the number of bytes sent or -1 if exits with some errors.

readData() : Receives a buffer where are present the message body items. Requires the identifier of file descriptor and pointer to a message body where save the data received. Returns the number of bytes read or -1 if exits with some errors.

sendData() : Sends a buffer where are present the message body items Requires the identifier of file descriptor and the pointer to a message body where saved the data to be sent. Returns the number of bytes sent or -1 if exits with some errors.

readMsg() : Receives a buffer where are present the message items. Requires the identifier of file descriptor and the pointer to a message where save the data that will be managed by server. Returns the number of bytes read or -1 if exits with some errors.

Function Documentation

int openConnection (char * *path*, unsigned int *ntimes*, unsigned int *secs*)

openConnection

opens a connection from client to server

Parameters:

path	path where located the socket
ntimes	number of reconnection attempts
secs	seconds that elapse between attempts

Returns:

sockfd identifier of file descriptor

int readData (long *fd*, message_data_t * *data*)

readData

Receives a buffer where are all body items to manage a request or to receive data from server Saves the items in a buffer and then copy them one at a time in the passed header. The receive is dived in two: first, receives the message body header second, receives the data of the message

Parameters:

connfd	Identifier of file descriptor
data	Pointer to the body of the message in which the received items will be saved

Returns:

size_buf Number of byte read or -1 if exits with error

int readHeader (long *connfd*, message_hdr_t * *hdr*)

readHeader

Receives a buffer where are all header items to know a request or to know the server response. Save the items in a buffer and then copy them one at a time in the passed header.

Parameters:

connfd	Identifier of file descriptor
hdr	Pointer to the header of the message in which the received items will be saved

Returns:

size_buf Number of byte read or -1 if exits with error

int readMsg (long *fd*, message_t * *msg*)

readMsg

Receives a request to the server. This request is a message with: code to define the request in the header and the other parts of the message to handle the request. Receives all items in a buffer and copies the items into the appropriate camp of the message passed.

The receive and copy are divided in three blocks: first, receives header items and copies them into the header of the message second, receives and then copies the message body header into the message passed third, receives directly the data of the body

Parameters:

connfd	Identifier of file descriptor
msg	Pointer to the message where located the request

Returns:

size_buf Number of byte received or -1 if exits with error

int sendData (long *fd*, message_data_t * *msg*)

sendData

Sends a buffer where are all body items to manage a request or to send data from server. Saves the items in a buffer coping them one at a time in the passed message body. The receive is dived in two: first, receives the message body header second, receives the data of the message

Parameters:

connfd	Identifier of file descriptor
msg	Pointer to the body of the message in which saved the items

Returns:

size_buf Number of byte sent or -1 if exits with error

int sendHeader (long *connfd*, message_hdr_t * *hdr*)

sendHeader

Function to send only message header.

Sends a buffer where are all header items to respond to the client. Copy the items one at a time from the passed header to a buffer and sends it.

Parameters:

connfd	Identifier of file descriptor
hdr	Pointer to the header of the message in which the received items will be saved

Returns:

size_buf Number of byte sent or -1 if exits with error

int sendRequest (long *fd*, message_t * *msg*)

sendRequest

Send a request to the server. This request is a message with: code to define the request in the header and, if necessary, the other parts of the message to handle the request. Copies all items in a buffer and send it. The copy and send are divided in three blocks: first, copies all items and send the header of the message second, sends the header of body of the message third, send directly the data of the body of the message.

Parameters:

connfd	Identifier of file descriptor
msg	Pointer to the message where located the request

Returns:

size_buf Number of byte sent or -1 if exits with error

Variable Documentation

int _MSGSIZE

_FILESIZE is the max size of a file could be sent - _MSGSIZE is the max size of a message could be sent

char Dir[256]

path where save the files received

char SOCKNAME[UNIX_PATH_MAX]

path where is saved the socket

hashtable.c File Reference

$hashtable_c$

Functions

- int hashVal (int key)
  hashVal
- hashtable_t * initTable (unsigned int length)
  initTable
- int insert (hashtable_t *table, char *name)
  insert
- user_t * search (hashtable_t *table, char *name)
  search
- int removing (hashtable_t *table, char *name)
  removing
- int addGroup (hashtable_t *table, char *name)
  addGroup
- group_chat_t * searchGroup (hashtable_t *table, char *name)
  searchGroup
- int removingGroup (hashtable_t *table, char *name)
  removingGroup

Variables

- int _MAX_HIST
  Variable used to save the size of users chat history.

Detailed Description

$hashtable_c$

LICENSE

Author:

Luca Canessa (516639) *

-

Copyright:

* Declares that all contents of this file are author's original operas *

-

DESCRIPTION

Looking overview

In this file there are functions to create and manipulate hash tables to register users. This hash table is designed as an array in which each cell represents an element in which a user and a group or more are saved. So one or more users or groups could be saved in an array cell. This is managed with a linked list. If a user U or a group G is the first to be added to cell C, it is saved directly to the cell, otherwise U or G is appended to the overflow queue C. ALL FUNCTIONS (except hashValue() and initTable()) MUST BE USED WITH HASH TABLE's LOCK ACQUIRED

hashValue() : Calculates the hash value using division method with table length Requires value of key: first letter of nickname of type integer Returns the hash value calculated

initTable() : Creates and initializes the hash table and its items. Requires the length of hash table of type UNISGNED INTEGER Return pointer to hash table created.

insert() : Inserts an user element in hash table. Requires pointer to hash table where insert user and pointer to user nickname Returns 1 if it has been added, otherwise 0.

search() : Search a user in hash table using its nickname Requires pointer to hash table where search and pointer to user nickname Returns pointer to user if found, else NULL.

removing() : Search an user in hash table and if found it, removes it. Requires pointer to the table and pointer to users nickname Returns 1 if it has been removed, otherwise 0

addGroup() : Allocates space to insert a group and initializes its items Requires pointer to the hash table and pointer to group name Returns 1 if the gruoup has been added, otherwise 0

searchGroup(): Search in the hash table a group using nickname. Requires pointer to the hash table and pointer to group's nickname. Returns pointer to the group found, otherwise NULL.

removingGroup(): Search the group with nickname passed and if found removes it. Requires pointer to the hash table and pointer to the group nickname. Returns 1 if it removed, otherwise 0.

Function Documentation

int addGroup (hashtable_t * *table*, char * *name*)

addGroup

Inserts an element in hash table using hash value calculated in 'hashValue' function. Creates a group and fills all its items. Initializes also its queue pointers: to chat hisoty and partecipants queue. If this group is the first in that cell then puts it directly in the table, else inserts it in the overflow queue.

Warning:

!!! IMPORTANT: acquire mutex before using this !!!

Parameters:

table	Pointer to hash table where insert the new element
name	Pointer to group name

Returns:

0/1 If user has been inserted then 1, else 0

int hashVal (int *key*)

hashVal

Given the key calculates hash value with division method using the table size

Parameters:

key	Value of key represented from the first character of name

Returns:

The hash value calculated

hashtable_t* initTable (unsigned int *length*)

initTable

Creates and initializes the hash table and set all parameters. Allocates space for users and groups. Allocates space also to save the online users

Parameters:

length	Size of hash table. It's used for users and group

Returns:

table Pointer to hash table just created

int insert (hashtable_t * *table*, char * *name*)

insert

Inserts an element in hash table using hash value calculated in 'hashValue' function. Creates an users and fills all its items. Initializes also its queue pointers: to chat hisoty and groups archive. If this user is the first in that cell then puts it directly in the table, else inserts it in the overflow queue.

Warning:

!!! IMPORTANT: acquire mutex before using this !!!

Parameters:

table	Pointer to hash table where insert the new element
name	Pointer to user name

Returns:

0/1 If user has been inserted then 1, else 0

int removing (hashtable_t * *table*, char * *name*)

removing

Search an user in hash table and if it found, it's removed. The operations to search user is the same of the function 'search()'. Instead, to remove the user, the function check if the user that must be removed is located directly in the table then check if exists an user into overflow queue, if it exists then swap the users, else remove the user. If the user to be removed is in the overflow queue then removes the node of queue.

Warning:

!!! IMPORTANT: acquire mutex before using !!!

Parameters:

table	Pointer to hash table
name	Pointer to user name 1/0 1 if the user has been removed, else 0

int removingGroup (hashtable_t * *table*, char * *name*)

removingGroup

Search a group in hash table and if it found, it's removed. The operations to search a group is the same of the function 'search()'. Instead, to remove the group, the function check if the group that must be removed is located directly in the table then check if exists a group into overflow queue, if it exists then swap the group, else remove the user. If the user to be removed is in the overflow queue then removes the node of queue.

Warning:

!!! IMPORTANT: acquire mutex before using !!!

Parameters:

table	Pointer to hash table
name	Pointer to group name 1/0 1 if the group has been removed, else 0

user_t* search (hashtable_t * *table*, char * *name*)

search

Search an user using its name passed as argument in hash table: if found it then return its pointer. To search, it starts from the cell with hash value calculated and then goes into the overflow queue.

Warning:

!!! IMPORTANT: acquire mutex before using !!!

Parameters:

table	Pointer to hash table where searching user
name	Pointer to user name to search

Returns:

Pointer to user found if exists, otherwise NULL

group_chat_t* searchGroup (hashtable_t * *table*, char * *name*)

searchGroup

Search a group using its name passed as argument in hash table: if found it then return its pointer. To search, it starts from the cell with hash value calculated and then goes into the overflow queue.

Warning:

!!! IMPORTANT: acquire mutex before using !!!

Parameters:

table	Pointer to hash table where searching user

name	Pointer to group name to search

Returns:

Pointer to group found if exists, otherwise NULL

Variable Documentation

int _MAX_HIST

Variable used to save the size of users chat history.

_MAX_HIST is number of history length

hashtable.h File Reference

$hashtable_h$

Data Structures

- struct us
- struct elem
- struct gr
- struct ht

Typedefs

- typedef struct us **user_t**
- typedef struct elem **ht_elem_t**
- typedef struct gr **ht_G_t**
- typedef struct ht **hashtable_t**

Functions

- hashtable_t * initTable (unsigned int length)
  initTable
- int insert (hashtable_t *table, char *name)
  insert
- user_t * search (hashtable_t *table, char *name)
  search
- int removing (hashtable_t *table, char *name)
  removing
- int addGroup (hashtable_t *table, char *name)
  addGroup
- group_chat_t * searchGroup (hashtable_t *table, char *name)
  searchGroup
- int removingGroup (hashtable_t *table, char *name)
  removingGroup

Variables

- int _MAX_HIST
  _MAX_HIST is number of history length

Detailed Description

$hashtable_h$

LICENSE

Author:

Luca Canessa (516639) *

-

Copyright:

* Declares that all contents of this file are author's original operas *

-

DESCRIPTION

Looking overview

This file contains prototypes and data structures to manage and manipulate a hash table in which ChatterBox's users and groups will be saved.

initTable() : Creates an hash table and initializes its items. Requires the length of the table. Returns th pointer to new hash table.

insert() : Adds an user into the hash table using his nickname. Requires the hash table pointer and the pointer to the user's name. Returns true if inserted, otherwise false.

search() : Search an user into the hash table and if found then return its pointer, else return NULL. Requires the hash table pointer and the pointer to the user's name. Returns the pointer to user if found, else NULL.

removing() : Search an user into the hash table and if found deletes it from table. Requires pointer to the hash table and pointer to user's nickname. Returns 1 if it exits without errors, else 0.

addGroup() : Allocates space to insert in hash table a chatting group. WARNING: USING WITH LOCK ACQUIRED Requires pointer to hash table and pointer to group's nickname. Returns 1 if the group is added, otherwise 0.

searchGroup(): Search in the hash table a group using nickname. Requires pointer to the hash table and pointer to group's nickname. Returns pointer to the group found, otherwise NULL.

removingGroup(): Search the group with nickname passed and if found removes it. Requires pointer to the hash table and pointer to the group nickname. Returns 1 if it removed, otherwise 0.

Function Documentation

int addGroup (hashtable_t * *table*, char * *name*)

addGroup

Inserts an element in hash table using hash value calculated in 'hashValue' function. Creates a group and fills all its items. Initializes also its queue pointers: to chat hisoty and partecipants queue. If this group is the first in that cell then puts it directly in the table, else inserts it in the overflow queue.

Warning:

!!! IMPORTANT: acquire mutex before using this !!!

Parameters:

table	Pointer to hash table where insert the new element
name	Pointer to group name

Returns:

0/1 If user has been inserted then 1, else 0

hashtable_t* initTable (unsigned int *length*)

initTable

Creates and initializes the hash table and set all parameters. Allocates space for users and groups. Allocates space also to save the online users

Parameters:

length	Size of hash table. It's used for users and group

Returns:

table Pointer to hash table just created

int insert (hashtable_t * *table*, char * *name*)

insert

Inserts an element in hash table using hash value calculated in 'hashValue' function. Creates an users and fills all its items. Initializes also its queue pointers: to chat hisoty and groups archive. If this user is the first in that cell then puts it directly in the table, else inserts it in the overflow queue.

Warning:

!!! IMPORTANT: acquire mutex before using this !!!

Parameters:

table	Pointer to hash table where insert the new element
name	Pointer to user name

Returns:

0/1 If user has been inserted then 1, else 0

int removing (hashtable_t * *table*, char * *name*)

removing

Search an user in hash table and if it found, it's removed. The operations to search user is the same of the function 'search()'. Instead, to remove the user, the function check if the user that must be removed is located directly in the table then check if exists an user into overflow queue, if it exists then swap the users, else remove the user. If the user to be removed is in the overflow queue then removes the node of queue.

Warning:

!!! IMPORTANT: acquire mutex before using !!!

Parameters:

table	Pointer to hash table
name	Pointer to user name 1/0 1 if the user has been removed, else 0

int removingGroup (hashtable_t * *table*, char * *name*)

removingGroup

Search a group in hash table and if it found, it's removed. The operations to search a group is the same of the function 'search()'. Instead, to remove the group, the function check if the group that must be removed is located directly in the table then check if exists a group into overflow queue, if it exists then swap the group, else remove the user. If the user to be removed is in the overflow queue then removes the node of queue.

Warning:

!!! IMPORTANT: acquire mutex before using !!!

Parameters:

table	Pointer to hash table
name	Pointer to group name 1/0 1 if the group has been removed, else 0

user_t* search (hashtable_t * *table*, char * *name*)

search

Search an user using its name passed as argument in hash table: if found it then return its pointer. To search, it starts from the cell with hash value calculated and then goes into the overflow queue.

Warning:

!!! IMPORTANT: acquire mutex before using !!!

Parameters:

table	Pointer to hash table where searching user
name	Pointer to user name to search

Returns:

Pointer to user found if exists, otherwise NULL

group_chat_t* searchGroup (hashtable_t * *table*, char * *name*)

searchGroup

Search a group using its name passed as argument in hash table: if found it then return its pointer. To search, it starts from the cell with hash value calculated and then goes into the overflow queue.

Warning:

!!! IMPORTANT: acquire mutex before using !!!

Parameters:

table	Pointer to hash table where searching user
name	Pointer to group name to search

Returns:

Pointer to group found if exists, otherwise NULL

Variable Documentation

int _MAX_HIST

_MAX_HIST is number of history length

_MAX_HIST is number of history length

---

message.h File Reference

Contiene il formato del messaggio.

Data Structures

- struct message_hdr_t
- struct message_data_hdr_t
- struct message_data_t
- struct message_t
- struct grp_msg

Typedefs

- typedef struct grp_msg **group_chat_t**

Functions

- static void setHeader (message_hdr_t *hdr, op_t op, char *sender)
  setheader
- static void setData (message_data_t *data, char *rcv, const char *buf, unsigned int len)
  setData

Detailed Description

Contiene il formato del messaggio.

Function Documentation

static void setData (message_data_t * *data*, char * *rcv*, const char * *buf*, unsigned int *len*)`[inline]`, `[static]`

setData

scrive la parte dati del messaggio

Parameters:

msg	puntatore al body del messaggio
rcv	nickname o groupname del destinatario
buf	puntatore al buffer
len	lunghezza del buffer

static void setHeader (message_hdr_t * *hdr*, op_t *op*, char * *sender*)`[inline], [static]`

setheader

scrive l'header del messaggio

Parameters:

hdr	puntatore all'header
op	tipo di operazione da eseguire
sender	mittente del messaggio

ops.h File Reference

Contiene i codici delle operazioni di richiesta e risposta.

Enumerations

- enum op_t { **REGISTER_OP** = 0, CONNECT_OP = 1, POSTTXT_OP = 2, POSTTXTALL_OP = 3, POSTFILE_OP = 4, GETFILE_OP = 5, GETPREVMSGS_OP = 6, USRLIST_OP = 7, UNREGISTER_OP = 8, DISCONNECT_OP = 9, CREATEGROUP_OP = 10, ADDGROUP_OP = 11, DELGROUP_OP = 12, OP_OK = 20, TXT_MESSAGE = 21, FILE_MESSAGE = 22, OP_FAIL = 25, OP_NICK_ALREADY = 26, OP_NICK_UNKNOWN = 27, OP_MSG_TOOLONG = 28, OP_NO_SUCH_FILE = 29, OP_ALREADY_ONLINE = 30, OP_NICK_AVAILABLE = 31, OP_GROUP_UNKNOWN = 32, OP_TOO_MANY_CONN = 33, OP_UNKNOWN = 34, OP_RM_USR_G = 35, GETPREVGROUPMSGS_OP = 36, MSG_NDELIV_SOMEUSERS = 37, **OP_END** = 100 }

Detailed Description

Contiene i codici delle operazioni di richiesta e risposta.

Enumeration Type Documentation

enum op_t

Enumerator:

CONNECT_OP	richiesta di registrazione di un ninckname
POSTTXT_OP	richiesta di connessione di un client
POSTTXTALL_OP	richiesta di invio di un messaggio testuale ad un nickname o groupname
POSTFILE_OP	richiesta di invio di un messaggio testuale a tutti gli utenti
GETFILE_OP	richiesta di invio di un file ad un nickname o groupname

GETPREVMSGS_OP	richiesta di recupero di un file
USRLIST_OP	richiesta di recupero della history dei messaggi
UNREGISTER_OP	richiesta di avere la lista di tutti gli utenti attualmente connessi
DISCONNECT_OP	richiesta di deregistrazione di un nickname o groupname
CREATEGROUP_OP	richiesta di disconnessione
ADDGROUP_OP	richiesta di creazione di un gruppo
DELGROUP_OP	richiesta di aggiunta ad un gruppo
OP_OK	richiesta di rimozione da un gruppo
TXT_MESSAGE	operazione eseguita con successo
FILE_MESSAGE	notifica di messaggio testuale
OP_FAIL	notifica di messaggio "file disponibile"
OP_NICK_ALREADY	generico messaggio di fallimento
OP_NICK_UNKNOWN	nickname o groupname gia' registrato
OP_MSG_TOOLONG	nickname non riconosciuto
OP_NO_SUCH_FILE	messaggio con size troppo lunga
OP_ALREADY_ONLINE	il file richiesto non esiste
OP_NICK_AVAILABLE	user with nickname already online
OP_GROUP_UNKNOWN	nickname already available in that group
OP_TOO_MANY_CONN	group not available
OP_UNKNOWN	too many connection in server
OP_RM_USR_G	unrecognized operation
GETPREVGROUPMSGS_OP	remove user from group
MSG_NDELIV_SOMEUSERS	richiesta di recupero della history dei messaggi di un gruppo

pool.c File Reference

$pool_c$

Macros

- #define **_POSIX_C_SOURCE** >= 199506L || _XOPEN_SOURCE >= 500

Functions

- void thread_work (threadpool_t *pool)
  thread_work
- threadpool_t * pool_creation ()
  pool_creation
- int threadpool_add (threadpool_t *pool, void(*functions)(void *), void *arg)
  threadpool_add
- int threadpool_free (threadpool_t *pool)
  threadpool_free
- int threadpool_destroy (threadpool_t *pool, int power_off)
  threadpool_destroy

Variables

- int _MAX_CONN
  _MAX_CONN is number of max connections to handle
- int _THREADn
  _THREADn is number of thread that operates in threadpool_t

Detailed Description

$pool_c$

LICENSE

Author:

Luca Canessa (516639) *

-

Copyright:

* Declares that all contents of this file are author's original operas *

-

DESCRIPTION

Looking overview

In this file there are body of functions to create and manipulate threads pool. The threads pool was designed like a table where are saved the pointer to array of ID threads, pointer to queue where will puts the tasks to be solved, mutual exclusion variable, condition variable, counter of tasks to do and a flag that determiny if threads shall be killed or shall be terminated after finished their works. The tasks scheduler was designed to assign one task at time per thread and after a thread finish its job, if the tasks queue is not empty takes the first element in head of queue, otherwise the condition variable puts scheduler in pause until that a task will be queued. So the scheduler is a FIFO queue.

thread_work() : If the counter of the activities in the queue is greater than zero, it extracts the activity (the function and its arguments) located in the head of the queue that must be executed by a thread, otherwise it is paused. It also updates the activity counter Requires pointer to a thread pool where is possible extract a task and its arguments. Returns nothing.

pool_creation(): Allocates space for new thread pool and initializes all its items. (It allocates space for threads, for tasks queue and start each threads required and initializes all variables). The number of threads in thread pool is taken from config file. Requires nothing Returns pointer to new thread pool.

threadpool_add():Adds a task (function and its arg) to queue, increases activity counter and unblocks held threads on conditional variable Requires pointer to thread pool, pointer to function and pointer to arg Returns 0 if terminate without errors, else -1

threadpool_free():Removes queued task, thread and then destroys pool Requires pointer to thread pool. Returns 0 if terminate without errors, else -1.

threadpool_destroy():Stops all worker threads, the stop mode depends on flag sent, then remove all threads and call 'threadpool_free()' to remove other elements. Requires pointer to thread pool Returns 0 if terminates without errors, else -1

Function Documentation

threadpool_t* pool_creation ()

pool_creation

It allocates space to create a new thread pool pf type 'threadpool_t'. It allocates space for threads and tasks, create threads and initializes all variables. To allocate space for tasks uses 'initialQueue()', to create threads uses 'malloc()' for IDs and 'pthread_create()' to create them. The number of thread in the pool is taken from config file from string 'ThreadsInPool'.

Returns:

pool Pointer to thread pool created.

void thread_work (threadpool_t * *pool*)

thread_work

It extracts from tasks queued the first, update the tasks counter and at end run the activity. The tasks represents function. This function keeps active until the thread where is located it keep alive

Parameters:

pool	Pointer to thread pool of type 'threadpool_t'

int threadpool_add (threadpool_t * *pool*, void(*)(void *) *functions*, void * *arg*)

threadpool_add

It adds a task (function and its arg) to queue, increases activity counter and unblocks held threads on conditional variable. (Some operations MUST done in safe condition, with mutex acquired)

Parameters:

pool	Pointer to the thread pool where add task
function	Pointer to the function to add
args	Pointer to the args of this function

Returns:

0 if it exits witout errors, -1 otherwise

int threadpool_destroy (threadpool_t * *pool*, int *power_off*)

threadpool_destroy

It stops all worker threads, the stop mode depends on flag sent, then remove all threads and call 'threadpool_free()' to remove other elements. Some operations MUST be executed with mutex acquired

Parameters:

pool	Pointer to the thread pool to be destroyed
power_off	Flag to stop worker, if 0 then stop immediately else stop when there are no more works

Returns:

err 0 if terminates without errors, else -1

int threadpool_free (threadpool_t * *pool*)

threadpool_free

It removes queued task, thread and then destroys pool. (Some operation MUST be done in safe mode, with mutex acquired)

Parameters:

pool	Pointer to the thread pool to be cleaned

Returns:

0 if it exits, -1 if 'pool' is NULL

Variable Documentation

int _MAX_CONN

_MAX_CONN is number of max connections to handle

max_conn is number of max connection to handle

int _THREADn

_THREADn is number of thread that operates in threadpool_t

num_thread is number of thread that operates in threadpool_t

pool.h File Reference

$pool_h$

Data Structures

- struct task
- struct pool

Typedefs

- typedef struct task **thread_task_t**
- typedef struct pool **threadpool_t**

Enumerations

- enum **threadpool_shutdown_t** { **immediate_shutdown** = 1, **graceful_shutdown** = 2 }
- enum **threadpool_destroy_flags_t** { **threadpool_graceful** = 1 }

Functions

- threadpool_t * pool_creation ()
  pool_creation
- int threadpool_add (threadpool_t *pool, void(*function)(void *), void *args)
  threadpool_add
- int threadpool_destroy (threadpool_t *pool, int flags)
  threadpool_destroy

Variables

- int _MAX_CONN
  max_conn is number of max connection to handle
- int _THREADn
  num_thread is number of thread that operates in threadpool_t

Detailed Description

$pool_h$

LICENSE

Author:

Luca Canessa (516639) *

-

Copyright:

* Declares that all contents of this file are author's original operas *

-

DESCRIPTION

Looking overview

This file contains prototypes of functions and data structures that manipulate and manage a thread pool. The thread pool is a structure in which some worker threads are present to perform tasks that are placed in a queue also present in this structure. To perform tasks, each thread has assigned a worker who extrapolates a job from the queue if present, otherwise it waits until someone inserts it. To create a pool you need to use 'pool_creation ()', to add jobs to the pool 'threadpool_add()' and to delete a pool you need to use 'threadpool_destroy ()'

pool_creation() : Allocates space for a new thread pool then returns pointer to new pool. Requires nothing Returns the pointer to the new thread pool

threadpool_add() : Adds a new job to the queue to be executed Requires a pointer to the thread pool to add the task and the function pointer and the related arguments to add to the queue Returns 0 if exits without errors, otherwise -1

threadpool_destroy():Stops all threads worker according to the mode sent, it can delete threads after they finished theri job or delete immediately. Requires pointer to the thread pool Returns 0 if terminate, else -1

Function Documentation

threadpool_t* pool_creation ()

pool_creation

Creates and initializes a thread pool

Returns:

Pointer to thread pool created or NULL

It allocates space to create a new thread pool pf type 'threadpool_t'. It allocates space for threads and tasks, create threads and initializes all variables. To allocate space for tasks uses 'initialQueue()', to create threads uses 'malloc()' for IDs and 'pthread_create()' to create them. The number of thread in the pool is taken from config file from string 'ThreadsInPool'.

Returns:

pool Pointer to thread pool created.

int threadpool_add (threadpool_t * *pool*, void(*)(void *) *functions*, void * *arg*)

threadpool_add

Adds task at thread pool task queue

Parameters:

pool	Pointer to 'threadpool_t' thread pool where add the task
function	Pointer to the function to add to the queue
args	Pointer to the args of job to add to queue

Returns:

0 if terminates without error -1 otherwise

It adds a task (function and its arg) to queue, increases activity counter and unblocks held threads on conditional variable. (Some operations MUST done in safe condition, with mutex acquired)

Parameters:

pool	Pointer to the thread pool where add task
function	Pointer to the function to add
args	Pointer to the args of this function

Returns:

0 if it exits witout errors, -1 otherwise

int threadpool_destroy (threadpool_t * *pool*, int *power_off*)

threadpool_destroy

Function to stop worker thread and destroy all pool

Parameters:

pool	Pointer to the pool to destroy
power_off	Flag to stop worker if 0 stops immediately, otherwise it stops when there are no more jobs

Returns:

0 if terminates without errors, else -1

It stops all worker threads, the stop mode depends on flag sent, then remove all threads and call 'threadpool_free()' to remove other elements. Some operations MUST be executed with mutex acquired

Parameters:

pool	Pointer to the thread pool to be destroyed
power_off	Flag to stop worker, if 0 then stop immediately else stop when there are no more works

Returns:

err 0 if terminates without errors, else -1

Variable Documentation

int _MAX_CONN

max_conn is number of max connection to handle

max_conn is number of max connection to handle

int _THREADn

num_thread is number of thread that operates in threadpool_t

num_thread is number of thread that operates in threadpool_t

queue.c File Reference

$queue_c$

Functions

- queue_t * initialQueue ()

initialQueue

- int push (queue_t *q, void *new_data)
  push
- void * pull (queue_t *q)
  pull
- int remove_node (queue_t *q, node_t *rm)
  remove_node
- void clear_queue (queue_t *q)
  clear_queue
- void destroy_queue (queue_t *q)
  destroy_queue

Detailed Description

$queue_c$

LICENSE

Author:

Luca Canessa (516639) *

-

Copyright:

* Declares that all contents of this file are author's original operas *

-

DESCRIPTION

Looking overview

This file contains functions used to manage queue structure. 'queue_t' is the type of double linked FIFO queue. This structure contains a pointer to head and another one to end of queue, both of type 'node_t', a counter of nodes and a mutex variable used to syncronizing the accesses.

initialQueue() : Allocates space for new queue of type queue_t and initializes its items. Returns pointer to new queue

push() : Creates a new node, fills its items, put node in the queue (ANY OPERATIONS INSIDE MUST BE SAFE WITH MUTEX ACQUIRED) Updates also the number of nodes inside the queue. Requires pointer to queue where put the node and pointer to element to add in the node. Returns 0 on success, -1 otherwise

pull() : Extracts the head node from queue and returns the pointer to element inside it. (ANY OPERATIONS INSIDE MUST BE SAFE WITH MUTEX ACQUIRED) Updates also the number of nodes inside the queue. Requires pointer to queue Returns the 'void *' pointer to the element extracted

remove_node() : Removes a particular node from queue BUT DOES NOT return the pointer to the element contained in this node. (ANY OPERATIONS INSIDE MUST BE SAFE

WITH MUTEX ACQUIRED) Update also the queue lenth. Requires the pointer to queue and pointer to the node to be removed. Returns 1 if successfully terminate, 0 otherwise.

clear_queue() : Removes all nodes presents in queue using 'pull()' function. This function uses 'free()' function on pointer returned from 'pull()'. (MAKE ATTENTION TO MEMORY LEAK) Requires pointer to queue to be cleaned. Returns nothing.

destroy_queue(): Cleans queue and frees memory using 'clear_queue()'. (Read 'clear_queue()' description for notes). Requires pointer to queue to be destroyed. Returns nothing.

Function Documentation

void clear_queue (queue_t * *q*)

clear_queue

Removes all nodes from queue and frees pointers returned from 'pull()'

Parameters:

q	Pointer to the queue to be cleaned

Bug:

Erases all node but not the last

void destroy_queue (queue_t * *q*)

destroy_queue

Removes all nodes from the queue and deletes the queue using 'clear_queue()'

Parameters:

q	Pointer to queue to clean.

queue_t* initialQueue ()

initialQueue

Allocates space for new queue, initializes its items. (allocates space also for first node to check later if element pointer exists)

Returns:

q Pointer to new queue if the space was allocated, NULL otherwise

void* pull (queue_t * *q*)

pull

Removes the node from head of queue and returns its element If the node to be removed is the only one, completely free the node and relocate it to return to the basic situation All operations MUST be doing in safe with mutual exclusion variables acquired

Parameters:

q	Pointer to queue

Returns:

ret (void *) pointer to element inside the node

int push (queue_t * *q*, void * *new_data*)

push

If insert the first element in queue, sets items of the existing node, else creates a new node and put it at end of the queue. In any case updates counter of nodes in queue. All operations MUST be doing in safe with mutual exclusion variables acquired

Parameters:

q	Pointer to the queue where add data
new_data	Pointer to data to add

Returns:

0 if terminates without errors, -1 otherwise.

int remove_node (queue_t * *q*, node_t * *rm*)

remove_node

Removes from queue the node pointed by rm. If the node to be removed is the only one, completely free the node and relocate it to return to the basic situation. All operations MUST be doing in safe with mutual exclusion variables acquired.

Parameters:

q	Pointer to queue
rm	Pointer to the node that needs to be removed

Returns:

1 if exit without errors, 0 otherwise

---

queue.h File Reference

$queue_h$

Data Structures

- struct node
- struct queue

Typedefs

- typedef struct node **node_t**
- typedef struct queue **queue_t**

Functions

- queue_t * initialQueue ()
  initialQueue
- int push (queue_t *q, void *new_data)
  push
- void * pull (queue_t *q)
  pull
- int remove_node (queue_t *q, node_t *rm)
  remove_node
- void clear_queue (queue_t *q)

clear_queue

- void destroy_queue (queue_t *q)
  destroy_queue

Detailed Description

$queue_h$

LICENSE

Author:

Luca Canessa (516639) *

-

Copyright:

 * Declares that all contents of this file are author's original operas *

-

DESCRIPTION

Looking overview

This file contains: the data structures that create the queues and the prototypes of the functions used to manage this structures. 'queue_t' is the type of double linked FIFO queue. This structure contains a pointer to head and another one to end of queue, both of type 'node_t', a counter of nodes and a mutex variable used to syncronizing the accesses.

initialQueue() : Allocates space to create a new queue and initializes its items. Does not require input arguments Returns pointer to new queue

push() : Creates a new node and puts it at the end of queue Requires pointer to queue where put the new item and pointer to the datus to insert Returns 0 if terminates with success, -1 otherwise

pull() : Extracts the head element from queue and returns the pointer to this element. Requires pointer to queue Returns the 'void *' pointer to the element extracted

remove_node() : Removes a particular node from queue BUT DOES NOT return the pointer to the element contained in this node. Requires the pointer to queue and pointer to the node to be removed. Returns 1 if successfully terminate, 0 otherwise.

clear_queue() : Removes all nodes presents in queue using 'pull()' function. This function uses 'free()' on node pointer. (MAKE ATTENTION TO MEMORY LEAK) Requires pointer to queue to be cleaned. Returns nothing.

destroy_queue() : Cleans queue and delete it. (Read 'clear_queue()' description for memory leak notes). Requires pointer to queue to be destroyed. Returns nothing.

Function Documentation

void clear_queue (queue_t * *q*)

clear_queue

Cleans queue removing all nodes

Warning:

!!! WARNING: this function remove all nodes in the queue and frees the element in nodes CAUTION to memory leak !!!

Parameters:

q	Pointer to the queue to be cleaned

Removes all nodes from queue and frees pointers returned from 'pull()'

Parameters:

q	Pointer to the queue to be cleaned

Bug:

Erases all node but not the last

void destroy_queue (queue_t * *q*)

destroy_queue

Clears all data from queue removing nodes and frees queue

Warning:

!!! WARNING: this function remove all nodes in the queue and frees the element in nodes CAUTION to memory leak !!!

Parameters:

q	pointer to queue to clean

Removes all nodes from the queue and deletes the queue using 'clear_queue()'

Parameters:

q	Pointer to queue to clean.

queue_t* initialQueue ()

initialQueue

Allocates space for new queue and initialize its elements

Returns:

Pointer to the created queue

Allocates space for new queue, initializes its items. (allocates space also for first node to check later if element pointer exists)

Returns:

q Pointer to new queue if the space was allocated, NULL otherwise

void* pull (queue_t * *q*)

pull

Extracts the first element from queue and returns it.

Parameters:

q	Pointer to the queue where extract the element.

Returns:

Returns pointer to element located in the node.

Removes the node from head of queue and returns its element If the node to be removed is the only one, completely free the node and relocate it to return to the basic situation All operations MUST be doing in safe with mutual exclusion variables acquired

Parameters:

q	Pointer to queue

Returns:

ret (void *) pointer to element inside the node

int push (queue_t * *q*, void * *new_data*)

push

Creates a new node, fills its items with new data and put node in queue.

Parameters:

q	Pointer to queue where insert the node.
new_data	Pointer to data to add.

Returns:

0 if function exit without errors, -1 otherwise.

If insert the first element in queue, sets items of the existing node, else creates a new node and put it at end of the queue. In any case updates counter of nodes in queue. All operations MUST be doing in safe with mutual exclusion variables acquired

Parameters:

q	Pointer to the queue where add data
new_data	Pointer to data to add

Returns:

0 if terminates without errors, -1 otherwise.

int remove_node (queue_t * *q*, node_t * *rm*)

remove_node

Removes from queue the node passed as parameter.

Warning:

!!! WARNING: this function remove node but NOT frees the element located in this node !!!

Parameters:

q	Pointer to the queue where remove the queue.
rm	Pointer to node to remove.

Returns:

1 if node has been removed, 0 otherwise

Removes from queue the node pointed by rm. If the node to be removed is the only one, completely free the node and relocate it to return to the basic situation. All operations MUST be doing in safe with mutual exclusion variables acquired.

Parameters:

q	Pointer to queue
rm	Pointer to the node that needs to be removed

Returns:

1 if exit without errors, 0 otherwise

signup.c File Reference

$signup_c$

Functions

- int checkin (hashtable_t *users, char *nick)
  checkin
- user_t * connecting (hashtable_t *users, char *nick)
  connecting
- int delete (hashtable_t *users, char *nick)
  delete

Detailed Description

$signup_c$

LICENSE

Author:

Luca Canessa (516639) *

-

Copyright:

* Declares that all contents of this file are author's original operas *

-

DESCRIPTION

Looking overview

In this file there are functions to manipulate user registering

checkin() : Inserts a nickname in hash table Requires a pointer to data structure where all users have been saved, in this case is an hash table designed in 'hashtable.h' file, and pointer to string where is the nickname to insert Returns '1' if it has been inserted into the table, '0' if exit with error, or 'OP_NICK_ALREADY' if user is already in the table

connecting() : Search an user in hash table in the hash table Requires a pointer to data structure (hash table designed in 'hashtable.h') and pointer to string where is nickname Returns a pointer to user of type 'user_t' if exists, else 'NULL'

delete() : Deletes an user from hash table Requires a pointer to hash table (designed in 'hashtable.h') where saved the user and the user's nickname of type char * Returns a bool value where 'true' represented by 1 and 'false' by 0

Function Documentation

int checkin (hashtable_t * *users*, char * *nick*)

checkin

Inserts a user of type 'user_t' in the hash table using an additional function called 'insert' if this user is not already present,

this checked with external function 'search'

Warning:

!!! WARNING : the functions 'search()' and 'insert()' inside of 'connecting()' MUST be protected with table mutex variables !!!

Parameters:

users	'hashtable_t' pointer to the hash table, where all users are saved
nick	Pointer of type 'char *' to string of user's nickname

Returns:

out Returns 1 if the user has been registered, 0 if it exits with some error, OP_NICK_ALREADY (26) if the user has already been registered previously

user_t* connecting (hashtable_t * *users*, char * *nick*)

connecting

Search an 'user_t' user with nickname 'nick' in the table using function 'search' declared in 'hashtable.h' file

Warning:

!!! WARNING : the function 'search()' inside 'connecting()' MUST be protected with table mutex variables !!!

Parameters:

users	Pointer of type 'hashtable_t' where all users are saved
nick	'char *' pointer to user with this 'nick' nickname

Returns:

user 'user_t' pointer that represent the user if it exists,'NULL' otherwise

int delete (hashtable_t * *users*, char * *nick*)

delete

Removes the 'user_t' user from hash table using external function 'removing' declared in 'hashtable.h', where has been declared

also the function 'search' used to know if this user exists

Warning:

!!! WARNING : the functions 'search()' and 'delete()' inside of 'connecting()' MUST be protected with table mutex variables !!!

Parameters:

users	Pointer of type 'hashtable_t' where all users are saved
nick	'char *' pointer to user with this 'nick' nickname

Returns:

out 1 if user has been deleted, 0 otherwise

signup.h File Reference

$signup_h$

Functions

- int checkin (hashtable_t *users, char *nick)
  checkin
- user_t * connecting (hashtable_t *users, char *nick)
  connecting
- int delete (hashtable_t *users, char *nick)
  delete

Detailed Description

$signup_h$

LICENSE

Author:
Luca Canessa (516639) *
-

Copyright:
* Declares that all contents of this file are author's original operas *
-

DESCRIPTION

Looking overview

In this file are presents the prototypes of functions to manage users registering

checkin() : Inserts a nickname in hash table Requires a pointer to data structure where all users have been saved, designed in 'hashtable.h' file, and pointer to string where is the nickname to insert Returns '1' if it has been inserted into the table, '0' if exit with error, or 'OP_NICK_ALREADY' if user is already in the table

connecting() : Search an user in hash table Requires a pointer to data structure (hash table designed in 'hashtable.h') and pointer to string where is nickname Returns a pointer to user of type 'user_t' if exists, else 'NULL'

delete() : Deletes an user from hash table Requires a pointer to hash table (designed in 'hashtable.h') where saved the user and the user's nickname of type char * Returns an int: 1 if the user has been deleted, else 0.

Function Documentation

int checkin (hashtable_t * *users*, char * *nick*)

checkin

Inserts a user with the nickname 'nick' in the hash table 'hashtable_t'

Parameters:

users	'hashtable_t' pointer to the hash table, where all users are saved
nick	Pointer of type 'char *' to string of user's nickname

Returns:

Returns 1 if the user has been registered, 0 if it exits with some error, OP_NICK_ALREADY (26) if the user has already been registered previously

Inserts a user of type 'user_t' in the hash table using an additional function called 'insert' if this user is not already present,

this checked with external function 'search'

Warning:

!!! WARNING : the functions 'search()' and 'insert()' inside of 'connecting()' MUST be protected with table mutex variables !!!

Parameters:

users	'hashtable_t' pointer to the hash table, where all users are saved
nick	Pointer of type 'char *' to string of user's nickname

Returns:

out Returns 1 if the user has been registered, 0 if it exits with some error, OP_NICK_ALREADY (26) if the user has already been registered previously

user_t* connecting (hashtable_t * *users*, char * *nick*)

connecting

Search an 'user_t' user with nickname 'nick' in the table

Parameters:

users	Pointer of type 'hashtable_t' where all users are saved
nick	'char *' pointer to user with this 'nick' nickname

Returns:

'user_t' pointer that represent the user if it exists,'NULL' otherwise

Search an 'user_t' user with nickname 'nick' in the table using function 'search' declared in 'hashtable.h' file

Warning:

!!! WARNING : the function 'search()' inside 'connecting()' MUST be protected with table mutex variables !!!

Parameters:

users	Pointer of type 'hashtable_t' where all users are saved
nick	'char *' pointer to user with this 'nick' nickname

Returns:

user 'user_t' pointer that represent the user if it exists,'NULL' otherwise

int delete (hashtable_t * *users*, char * *nick*)

delete

Removes if it exists the 'user_t' user from hash table

Parameters:

users	Pointer of type 'hashtable_t' where all users are saved
nick	'char *' pointer to user with this 'nick' nickname

Returns:

1 if user has been deleted, 0 otherwise

Removes the 'user_t' user from hash table using external function 'removing' declared in 'hashtable.h', where has been declared

also the function 'search' used to know if this user exists

Warning:

!!! WARNING : the functions 'search()' and 'delete()' inside of 'connecting()' MUST be protected with table mutex variables !!!

Parameters:

users	Pointer of type 'hashtable_t' where all users are saved
nick	'char *' pointer to user with this 'nick' nickname

Returns:

out 1 if user has been deleted, 0 otherwise

Index

INDEX