File scheduler

Generated by Doxygen 1.8.14

Thu Apr 12 2018 01:18:19

Contents

1 PYSSC scheduler.

This server is used to synchronize pssc workers that run at different nodes.

Main problem is that each of them have to know whether to generate 'reuse' file read it or wait while it is being generated by other worker(node).

This server recieves data about requested file and respond with 'suggestion': READ, WRIT, WAIT.

In short. If file does not exist and is not being generated - generate it (WRIT). If file exists - read it (READ). Read operation does not change the data, so can be done in parallel, anyway file will be cached into RAM. If file is being generated - WAIT for a next READ message.

Communication is done by epoll. It is possible to add any reasonable number of threads if needed, but for the environment it was codded for - two threads is more than enough. First thread accepts connections, second communicates with clients.

This server should be launched on one of the nodes. Other clients should know server's ip. Because of the asynchronous design, it produces very little overhead. When received SIGINT - all connection should be closed and program terminated. Check PYSSC git for a client version. Although this server was tested with many threads and for a long time, it may still have some error or space for improvement. I would be glad to hear any response.

Code sucessfully passedd PVS-Studio and Valgrind check.

2 Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

3 File Index

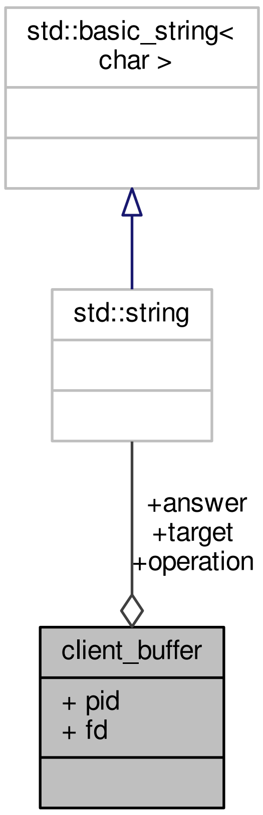
3.1 File List

Here is a list of all files with brief descriptions:

4 Data Structure Documentation

4.1 client_buffer Struct Reference

Collaboration diagram for client_buffer:



Data Fields

- int pid
- int fd
- string operation
- string target
- string answer

4.1.1 Detailed Description

Definition at line 92 of file file_scheduler.cpp.

4.1.2 Field Documentation

4.1.2.1 answer

```
string client_buffer::answer
```

Definition at line 98 of file file_scheduler.cpp.

4.1.2.2 fd

```
int client_buffer::fd
```

Definition at line 95 of file file_scheduler.cpp.

4.1.2.3 operation

```
string client_buffer::operation
```

Definition at line 96 of file file_scheduler.cpp.

4.1.2.4 pid

```
int client_buffer::pid
```

Definition at line 94 of file file_scheduler.cpp.

4.1.2.5 target

```
string client_buffer::target
```

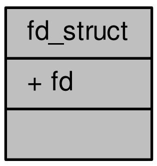
Definition at line 97 of file file_scheduler.cpp.

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

- file_scheduler.cpp

4.2 fd_struct Struct Reference

Collaboration diagram for fd_struct:



Data Fields

- int fd

4.2.1 Detailed Description

Definition at line 82 of file file_scheduler.cpp.

4.2.2 Field Documentation

4.2.2.1 fd

```
int fd_struct::fd
```

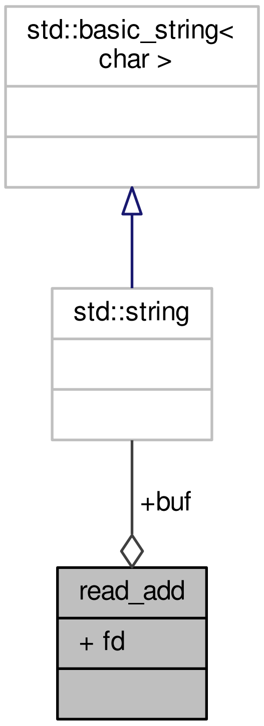
Definition at line 84 of file file_scheduler.cpp.

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

- file_scheduler.cpp

4.3 read_add Struct Reference

Collaboration diagram for read_add:



Data Fields

- int fd
- string buf

4.3.1 Detailed Description

Definition at line 101 of file file_scheduler.cpp.

4.3.2 Field Documentation

4.3.2.1 buf

```
string read_add::buf
```

Definition at line 104 of file file_scheduler.cpp.

4.3.2.2 fd

```
int read_add::fd
```

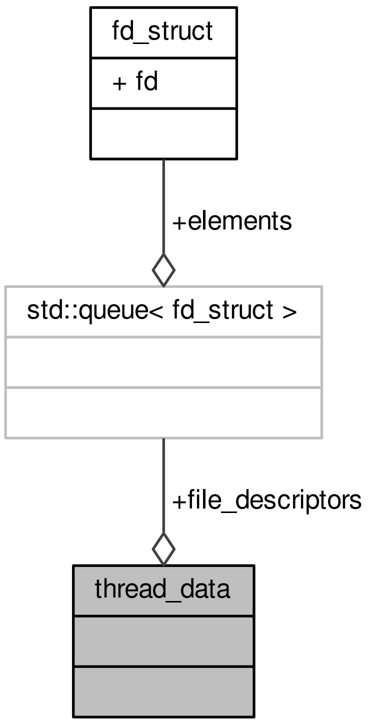
Definition at line 103 of file file_scheduler.cpp.

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

- file_scheduler.cpp

4.4 thread_data Struct Reference

Collaboration diagram for thread_data:



Data Fields

- queue< fd_struct > file_descriptors

4.4.1 Detailed Description

Definition at line 87 of file file_scheduler.cpp.

4.4.2 Field Documentation

4.4.2.1 file_descriptors

```
queue<fd_struct> thread_data::file_descriptors
```

Definition at line 89 of file file_scheduler.cpp.

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

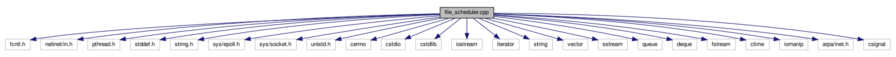
- file_scheduler.cpp

5 File Documentation

5.1 file_scheduler.cpp File Reference

```
#include <fcntl.h>
#include <netinet/in.h>
#include <pthread.h>
#include <stddef.h>
#include <string.h>
#include <sys/epoll.h>
#include <sys/socket.h>
#include <unistd.h>
#include <cerrno>
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
#include <sstream>
#include <queue>
#include <deque>
#include <fstream>
#include <ctime>
#include <iomanip>
#include <arpa/inet.h>
#include <csignal>
```

Include dependency graph for file_scheduler.cpp:



Data Structures

- struct fd_struct
- struct thread_data
- struct client_buffer
- struct read_add

Macros

- #define MAXEVENTS 500

Functions

- size_t parse_buffer (string str, deque< client_buffer > *client_buf, int fd)
- int secure_send (client_buffer *client_buf)
- void check_client_errors (deque< client_buffer > *processed_client_buf, deque< client_buffer > *client_↩ buf, const ssize_t fd)
- void * read_and_respond (void *threadarg)
- int accept_connections (uint16_t port, queue< fd_struct > *clients)
- void signalHandler (int signum)
- int main ()

Variables

- pthread_mutex_t lock
- bool time_to_exit = false
- int exit_code = 0

5.1.1 Macro Definition Documentation

5.1.1.1 MAXEVENTS

```
#define MAXEVENTS 500
```

Definition at line 79 of file file_scheduler.cpp.

5.1.2 Function Documentation

5.1.2.1 accept_connections()

```
int accept_connections (
            uint16_t port,
            queue< fd_struct > * clients )
```

Initializes socket on port 'port' and waits for connections. On incomming connection sets associated socket to async mode and stores in fd_struct structure which is shared with processing thread.

Parameters

in	*port*	Port used to create listening socket.
in	*clients*	queue that stores file descriptors(sockets) accepted on port 'port'.

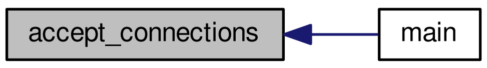
Returns

status code

Definition at line 668 of file file_scheduler.cpp.

```
{
    std::string rcv;
    int listen_fd, comm_fd;
    struct sockaddr_in servaddr;
    std::ofstream log_main;
    log_main.open("incoming.log", std::ios::out | std::ios::app);
    #ifdef DEBUG
        auto t = time(nullptr);
        auto tm = *localtime(&t);
        log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Thread created\n";
    #endif

    listen_fd = socket(AF_INET, SOCK_STREAM, 0);

    if (listen_fd == -1)
    {
        cout << "Can't create file descriptor." << endl;
        exit_code = 1;
        time_to_exit = true;
        sleep(10);
        exit(1);
    }

    memset( &servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htons(INADDR_ANY);
    servaddr.sin_port = htons(port);
    #ifdef DEBUG
        t = time(nullptr);
        tm = *localtime(&t);
        log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Attempting to listen on " << port << " port\n
";
    #endif
    int my_timer = 20;

    while(my_timer > 0)
    {
        if(bind(listen_fd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0)
            sleep(10);
        else
            break;
        my_timer--;
    }

    if(my_timer == 0)
    {
        cout << "Binding to socket error." << endl;
        exit_code = 1;
        time_to_exit = true;
        sleep(10);
        exit(2);
    }
    #ifdef DEBUG
        t = time(nullptr);
        tm = *localtime(&t);
        log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Successful bind to the socket\n";
    #endif

    fd_struct temp;

    while(!time_to_exit)
    {
        listen(listen_fd, 60);
        #ifdef DEBUG
            t = time(nullptr);
            tm = *localtime(&t);
            log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Waiting for incoming connections.\n";
            log_main.flush();
        #endif
        sockaddr_in clientAddr;
        socklen_t sin_size=sizeof(struct sockaddr_in);
        comm_fd = accept(listen_fd, (struct sockaddr*)&clientAddr, &sin_size);

        if(comm_fd == -1)
        {
            cout << "Connection acceptance error." << endl;
            //      exit(3);
        }

        #ifdef DEBUG
            t = time(nullptr);
            tm = *localtime(&t);
            char loc_addr[INET_ADDRSTRLEN+1];
            inet_ntop(AF_INET, &(clientAddr.sin_addr), loc_addr, INET_ADDRSTRLEN);
            log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Incoming connection on descriptor " << comm_fd << " from " << loc_addr << ":" << clientAddr.sin_port <<"\n";
        #endif
        //      make nonblocking
        auto flags = fcntl (comm_fd, F_GETFL, 0);
        if (flags < 0)
        {
            perror ("fcntl");
            return -1;
        }

        flags |= O_NONBLOCK;
        auto s = fcntl (comm_fd, F_SETFL, flags);

        if(s < 0)
        {
            perror ("fcntl");
            time_to_exit = true;
            log_main.close();
            return -1;
        }
        #ifdef DEBUG
            t = time(nullptr);
            tm = *localtime(&t);
            log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Switched " << comm_fd <<" to nonblocking mode.\n";
        #endif

        temp.fd = comm_fd;

        pthread_mutex_lock(&lock);
            clients->push(temp);
        pthread_mutex_unlock(&lock);
        #ifdef DEBUG
            t = time(nullptr);
            tm = *localtime(&t);
            log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Successfully pushed into queue for further processing.\n";
        #endif
    }

    #ifdef DEBUG
    log_main.close();
    #endif

    return 0;
}
```

Here is the caller graph for this function:



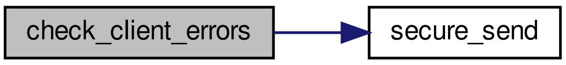
5.1.2.2 check_client_errors()

```
void check_client_errors (
            deque< client_buffer > * processed_client_buf,
            deque< client_buffer > * client_buf,
            const ssize_t fd )
```

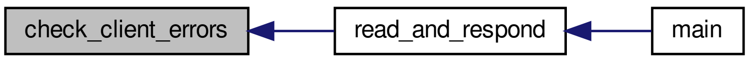
Definition at line 208 of file file_scheduler.cpp.

```
{
    string error_target = "";

    for(auto iter = client_buf->begin(); iter != client_buf->end(); ++iter)
    {
        if(iter->fd == fd)
            return;
    }

    for(auto iter = processed_client_buf->begin(); iter != processed_client_buf->end(); ++iter)
    {
        if(iter->fd == fd )
        {
            error_target = iter->target;
            cerr << "Broken client removing: " << iter->fd << " " << iter->operation << " " << iter->target
       << endl;
            processed_client_buf->erase(iter);
            break;
        }
    }
    if(error_target.length() > 0)
    {
        for(auto iter = processed_client_buf->begin(); iter != processed_client_buf->end(); ++iter)
        {
            if(iter->target == error_target && iter->answer == "WAIT")
            {
                iter->answer = "WRIT";
                cerr << "PID " << iter->pid << " advised to WRIT";
                if(secure_send(&*iter) != 0)
                    cerr << "ERROR in secure send";
                break;
            }
        }
    }
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.2.3 main()

```
int main ( )
```

Nothing fancy. Creates a thread and launches connection accepting function

Returns

status code to OS

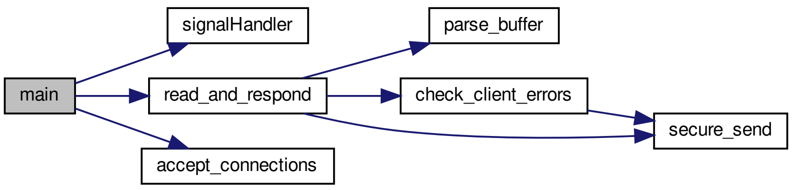
Parameters

clients	data structure to store fd

Definition at line 802 of file file_scheduler.cpp.

```
{
    queue <fd_struct> clients;
    pthread_t threads[1];

    signal(SIGINT, signalHandler);

    if (pthread_mutex_init(&lock, NULL) != 0)
    {
        printf("\n mutex init failed\n");
        return 1;
    }

    int rc = pthread_create(&threads[0], NULL, read_and_respond, (void *)&clients);
    if (rc)
    {
        cout << "Error:unable to create thread," << rc << endl;
        exit_code = -1;
        time_to_exit = true;
        sleep(10);
        return exit_code;

    }

    accept_connections(1987, &clients);

    pthread_join(rc, NULL);
    pthread_mutex_destroy(&lock);

    return exit_code;
}
```

Here is the call graph for this function:



5.1.2.4 parse_buffer()

```
size_t parse_buffer (
            string str,
            deque< client_buffer > * client_buf,
            int fd )
```

Parses input buffer and stores parsed messages in queue It may parse more than one message(stored in str) and if last massage is incomplete - returns how many characters to save in external buffer for future processing.

Parameters

in	*str*	Input buffer with data recieved in socket fd.
in	*client_buf*	Structure that keeps all parsed messages.
in	*fd*	file descriptor associated with passed buffer data.

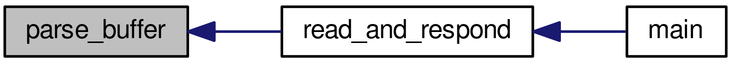
Returns

how many characters to save in external buffer for future processing

Definition at line 136 of file file_scheduler.cpp.

```
{
    bool done = false;
    bool error = false;
    client_buffer temp;
    vector <string> info_block;
    string current_str;
    size_t found = 0;
    string token;

    while(str.length() > 0 && !done && !error)
    {
        found = str.find_first_of("#");
        if(found == std::string::npos)
        {
            error = true;
            continue;
        }

        unsigned fut_len = stoi(str.substr(0, found));
        if(str.length() < found + 1 + fut_len)
            break;

        current_str = str.substr(found+1, fut_len);
        str = str.substr(found + 1 + fut_len);

        std::istringstream iss(current_str);
        info_block.clear();
        while (getline(iss, token, '#'))
            info_block.push_back(token);

        current_str.clear();

        temp = {};
        if(info_block.size() > 0 )
            temp.pid = stoi(info_block[0]);

        if(info_block.size() > 1 )
            temp.operation = info_block[1];

        if(info_block.size() > 2 )
            temp.target = info_block[2];

        temp.fd = fd;

        if(temp.operation == "DONE")
            client_buf->push_front(temp);
        else
            client_buf->push_back(temp);

        if(str.length() < 3)
            done = true;
    }

    return str.length();
}
```

Here is the caller graph for this function:



5.1.2.5 read_and_respond()

```
void * read_and_respond (
            void * threadarg )
```

Registers new sockets in epoll function(waits on data in async mode). Reads data from sockets in async mode. Calls parse function and makes decision according to the processed requests.

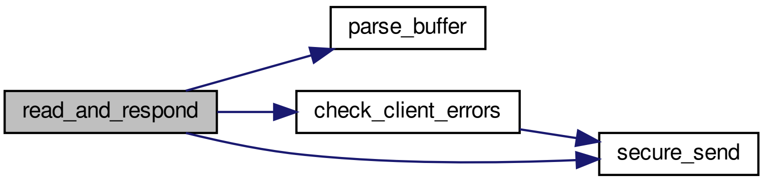
Parameters

in	*threadarg*	Structure that contains address of queue with file descriptors.

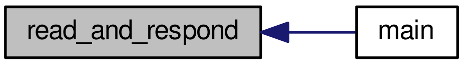
Definition at line 248 of file file_scheduler.cpp.

```
{
    auto my_data = (thread_data *) threadarg;
    queue <fd_struct> *fds = &my_data->file_descriptors;
    struct epoll_event event;
    struct epoll_event *events;
    auto efd = epoll_create1 (0);
    vector <int> fd_to_remove;
//  vector <size_t> local_fd;
    deque <client_buffer> client_buf;
    deque <client_buffer> processed_client_buf;
    vector <read_add> buff_add;
    read_add ra;
    events = (epoll_event*)calloc (MAXEVENTS, sizeof event);
    event.events = EPOLLIN | EPOLLET;
    size_t fd_event_counter = 0;
    int n;
    std::ofstream log_processing;
    log_processing.open("processing.log", std::ios::out | std::ios::app);
    #ifdef DEBUG
        auto t = time(nullptr);
        auto tm = *localtime(&t);
        log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Thread created\n";
    #endif

    while(!time_to_exit)
    {
        #ifdef DEBUG
            if(fd_to_remove.size() > 0)
            {
                t = time(nullptr);
                tm = *localtime(&t);
                log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Descriptors to clean: " <<
fd_to_remove.size() << "\n";
            }
        #endif
        for(unsigned j=0; j < fd_to_remove.size(); ++j)
        {
            for(size_t k = 0; k < buff_add.size(); ++k)
            {
                if(buff_add[k].fd == fd_to_remove[j])
                {
                    #ifdef DEBUG
                        t = time(nullptr);
                        tm = *localtime(&t);
                        log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Removing " << fd_to_remove[j] << endl;
                    #endif
                    buff_add.erase(buff_add.begin() + k);
                    break;
                }
            }

            for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end();)
            {
                if(iter->fd == fd_to_remove[j])
                    iter = processed_client_buf.erase(iter);
                else
                    iter++;
            }

        }

        fd_event_counter -= fd_to_remove.size();
        fd_to_remove.clear();

        #ifdef DEBUG
            if(!fds->empty())
            {
                t = time(nullptr);
                tm = *localtime(&t);
                log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Processing new connections\n";
                if(!fds->empty())
                    log_processing.seekp(-1, std::ios_base::cur);
            }
        #endif

        while(!fds->empty())
        {
            pthread_mutex_lock(&lock);
            event.data.fd = fds->front().fd;
            fds->pop();
            pthread_mutex_unlock(&lock);
            ++fd_event_counter;
            #ifdef DEBUG
                log_processing << ".";
            #endif

            if( epoll_ctl(efd, EPOLL_CTL_ADD, event.data.fd, &event) == -1)
            {
                perror ("epoll_ctl");
                cerr << "EPOLL ERROR\n";
                time_to_exit = true;
                #ifdef DEBUG
                    log_processing.seekp(0, std::ios_base::end);
                    t = time(nullptr);
                    tm = *localtime(&t);
                    log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "EPOLL ERROR\n";
                    log_processing.close();
                #endif
                pthread_exit(NULL);
            }
        }

        #ifdef DEBUG
            log_processing.seekp(0, std::ios_base::end);
        #endif

        if(fd_event_counter > 0)
        {
            n = epoll_wait(efd, events, MAXEVENTS, 1000);
            for(int i = 0; i < n; ++i)
            {
                if (( &events[i] != NULL) && ((events[i].events & EPOLLERR) || (events[i].events & EPOLLHUP) ||  (!(events[i].events & EPOLLIN))))
                {
                    cerr << "epoll error\n";
                    close (events[i].data.fd);
                    fd_to_remove.push_back(events[i].data.fd);
                    continue;
                }
                else
                {
                    int done = 0;
                    ra.buf = "";

                    for(unsigned k = 0; k < buff_add.size(); ++k)
                    {
                        if(buff_add[k].fd == events[i].data.fd)
                        {
                            ra.buf = buff_add[k].buf;
                            break;
                        }
                    }
                    #ifdef DEBUG
                        log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Reading from socket " << events[i].data.fd << endl;
                    #endif
                    while (1)
                    {
                        ssize_t count;
                        char buf[512];
                        memset(buf,0, sizeof buf);
                        count = recv(events[i].data.fd, buf, sizeof buf, 0);
                        #ifdef DEBUG
                                t = time(nullptr);
                                tm = *localtime(&t);
                                log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Recieved: " << string(buf) << endl;
                        #endif
                        if (count == -1)
                        { // If errno == EAGAIN, that means we have read all data. So go back to the main loop.
                            if (errno != EAGAIN)
                            {
                                perror ("read");
                                cerr << "Count error";
                                #ifdef DEBUG
                                t = time(nullptr);
                                tm = *localtime(&t);
                                log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "ERROR: Count error in epoll" << endl;
                                #endif
                                done = 1;
                            }
                            break;
                        }
                        else if (count == 0)
                        { // End of file. The remote has closed the connection.
                            done = 1;
                            break;
                        }
                        ra.buf += string(buf);
                        #ifdef DEBUG
                            t = time(nullptr);
                            tm = *localtime(&t);
                            log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Parsing message of length " << ra.buf.length() << endl;
                            log_processing << "Message: " << string(ra.buf) << endl;
                        #endif
                        size_t char_left = parse_buffer(ra.buf, &client_buf, events[i].data.fd);
                        #ifdef DEBUG
                        t = time(nullptr);
                        tm = *localtime(&t);
                        if(client_buf.size() > 0)
                        {
                            for(auto iter = client_buf.begin(); iter != client_buf.end(); ++iter)
                            {
                                log_processing << "PID " << iter->pid << " on " << iter->fd << " requested " << iter->operation << " " << iter->target << endl;
                            }
                        }
                        else
                            log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "WARNING: Client buffer is empty\n";

                        #endif
                        if(char_left > 0)
                        {
                            #ifdef DEBUG
                                t = time(nullptr);
                                tm = *localtime(&t);
                                log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Not all information was received. Missing " << char_left << " chars\n";
                            #endif

                            ra.fd = events[i].data.fd;
                            unsigned k = 0;
                            for(; k < buff_add.size(); ++k)
                            {
                                if(buff_add[k].fd == events[i].data.fd)
                                {
                                    buff_add[k].buf += ra.buf;
                                    break;
                                }
                            }
                            if(k == buff_add.size())
                                buff_add.push_back(ra);
                        }
                        else
                        {
                            #ifdef DEBUG
                                t = time(nullptr);
                                tm = *localtime(&t);
                                log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Success in reading full message\n";
                            #endif
                            for(unsigned k = 0; k < buff_add.size(); ++k)
                            {
                                if(buff_add[k].fd == events[i].data.fd)
                                {
                                    buff_add.erase(buff_add.begin() + k);
                                    break;
                                }
                            }
                        }
                    }

                    if (done)
                    {
                        #ifdef DEBUG
                        t = time(nullptr);
                        tm = *localtime(&t);
                        log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Closing connection on descriptor " << events[i].data.fd << endl;
                        #endif

                        check_client_errors(&processed_client_buf, &client_buf, events[i].data.fd);

                        close (events[i].data.fd); // Closing the descriptor will make epoll remove it from the set of descriptors which are monitored.
                        fd_to_remove.push_back(events[i].data.fd);
                    }
                }
            }

            #ifdef DEBUG
            if(!client_buf.empty())
            {
                t = time(nullptr);
                tm = *localtime(&t);
                log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Begin processing requests.\n";
            }
            #endif

            #ifdef DEBUG
            log_processing << "processed_client_buf before processing new requests \n************\n";
            for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end(); ++iter)
            {
                log_processing << "PID " << iter->pid << " from socket " << iter->fd << " requested " << iter->operation << " " << iter->target << " advised " << iter->answer << endl;
            }
            log_processing << "************\n end\n";
            #endif

            while(!client_buf.empty())
            {
                if(client_buf.front().operation == "READ")
                {
                    #ifdef DEBUG
                        t = time(nullptr);
                        tm = *localtime(&t);
                        log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "PID: " << client_buf.front().pid << " from socket " << client_buf.front().fd << " wants to read " << client_buf.front().target << endl;
                    #endif
                    client_buf.front().answer = "READ";

                    for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end(); ++iter)
                    {
                        if(iter->target == client_buf.front().target )
                        {
                            if(iter->answer == "WRIT" || iter->answer == "WAIT")
                            {
                                client_buf.front().answer = "WAIT";
                                break;
                            }
                            else if(iter->answer == "READ")
                            {
                                client_buf.front().answer = "READ";
                                break;
                            }
                        }
                    }

                    if(secure_send(&client_buf.front()) == 0)
                        processed_client_buf.push_back(client_buf.front());
                    else
                        cerr << "ERROR in secure send";

                    #ifdef DEBUG
                        t = time(nullptr);
                        tm = *localtime(&t);
                        log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "RESPONSE to PID: " << client_buf.front().pid << " from socket " << client_buf.front().fd << ": " << client_buf.front().answer << endl;
                    #endif
                }
                else if(client_buf.front().operation == "WRIT")
                {
                    #ifdef DEBUG
                        t = time(nullptr);
                        tm = *localtime(&t);
                        log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "PID: " << client_buf.front().pid << " from socket " << client_buf.front().fd << " wants to write " << client_buf.front().target << endl;
                    #endif
                    client_buf.front().answer = "WRIT";
                    for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end(); ++iter)
                    {
                        if(iter->target == client_buf.front().target )
                        {
                            if(iter->answer == "WRIT" || iter->answer == "WAIT")
                            {
                                client_buf.front().answer = "WAIT";
                                break;
                            }
                            else if(iter->answer == "READ")
                            {
                                client_buf.front().answer = "READ";
                                break;
                            }
                        }
                    }

                    if(secure_send(&client_buf.front()) == 0)
                        processed_client_buf.push_back(client_buf.front());
                    else
                        cerr << "ERROR in secure send";

                    #ifdef DEBUG
                        t = time(nullptr);
                        tm = *localtime(&t);
                        log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "RESPONSE to PID: " << client_buf.front().pid << " from socket " << client_buf.front().fd << ": " << client_buf.front().answer << endl;
                    #endif
                }
                else if(client_buf.front().operation == "DONE")
                {
                    #ifdef DEBUG
                        t = time(nullptr);
                        tm = *localtime(&t);
                        log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "PID: " << client_buf.front().pid << " from socket " << client_buf.front().fd << " finished working with " << client_buf.front().target << endl;
                    #endif

                    for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end();)
                    {
                        if(iter->target == client_buf.front().target )
                        {
                            if(iter->pid == client_buf.front().pid)
                            {
                                #ifdef DEBUG
                                    t = time(nullptr);
                                    tm = *localtime(&t);
                                    log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "PID: " << iter->pid << " from socket " << iter->fd << ": is DONE - SELF-DESTRUCTION" << endl;
                                #endif
                                iter = processed_client_buf.erase(iter);
                                continue;
                            }
                            else if(iter->answer == "WAIT")
                            {
                                iter->answer = "READ";
                                if(secure_send(&*iter) != 0)
                                    cerr << "ERROR in secure send";

                                #ifdef DEBUG
                                    t = time(nullptr);
                                    tm = *localtime(&t);
                                    log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "RESPONSE to PID: " << iter->pid << " from socket " << iter->fd << ": " << iter->answer << endl;
                                #endif
                            }
                        }

                        ++iter;
                    }
                }
                else
                    cerr << client_buf.front().operation << endl;

                client_buf.pop_front();
            }

            #ifdef DEBUG
            log_processing << "processed_client_buf after processing new requests \n************\n";
            for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end(); ++iter)
            {
                log_processing << "PID " << iter->pid << " from socket " << iter->fd << " requested " << iter->operation << " " << iter->target << " advised " << iter->answer << endl;
            }
            log_processing << "************\n end\n";
            #endif
        }
        else
            sleep(1);
    }

    for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end();)
    {
        iter->answer = "EXIT";
//      cerr <<
        secure_send(&*iter);
        close(iter->fd);
    }
    processed_client_buf.clear();

    #ifdef DEBUG
        log_processing.seekp(0, std::ios_base::end);
        t = time(nullptr);
        tm = *localtime(&t);
        log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Time to exit\n";
        log_processing.close();
    #endif

    pthread_exit(NULL);
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



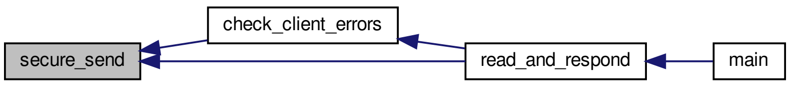
5.1.2.6 secure_send()

```
int secure_send (
             client_buffer * client_buf )
```

Definition at line 193 of file file_scheduler.cpp.

```
{
    for(size_t as = 0; as < client_buf->answer.length();)
    {
        auto sent = send(client_buf->fd, client_buf->answer.substr(as).c_str(), client_buf->
      answer.substr(as).length(), 0);
        if(sent < 0)
        {
            cerr << "Error on socket " << client_buf->fd << endl;
            return -1;
        }
        as += sent;
    }
    return 0;
}
```

Here is the caller graph for this function:



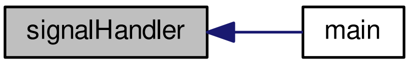
5.1.2.7 signalHandler()

```
void signalHandler (
             int signum )
```

Definition at line 117 of file file_scheduler.cpp.

```
{
    cerr << "Interrupt signal (" << signum << ") received.\n";
    cerr << "Programm has 30 seconds to finish or it will be forced to exit.\n";
    exit_code = signum;
    time_to_exit = true;
    sleep(30);
    cerr << "Looks like program does not respond. Killing.\n";
    exit(signum);
}
```

Here is the caller graph for this function:



5.1.3 Variable Documentation

5.1.3.1 exit_code

```
int exit_code = 0
```

Definition at line 115 of file file_scheduler.cpp.

5.1.3.2 lock

```
pthread_mutex_t lock
```

Definition at line 113 of file file_scheduler.cpp.

5.1.3.3 time_to_exit

```
bool time_to_exit = false
```

Definition at line 114 of file file_scheduler.cpp.

5.2 file_scheduler.cpp

```
// This is a personal academic project. Dear PVS-Studio, please check it.
// PVS-Studio Static Code Analyzer for C, C++ and C#: http://www.viva64.com
/** @file file_scheduler.cpp*/
/** This server is used to synchronize pssc workers that run at different nodes.
**
** Main problem is that each of them have to know whether to generate 'reuse' file
** read it or wait while it is being generated by other worker(node).
**
** This server recieves data about requested file and respond with 'suggestion':
** READ, WRIT, WAIT.
**
** In short. If file does not exist and is not being generated - generate it (WRIT).
** If file exists - read it (READ). Read operation does not change the data, so can be
** done in parallel, anyway file will be cached into RAM.
** If file is being generated - WAIT for a next READ message.
**
** Communication is done by epoll. It is possible to add any reasonable number
** of threads if needed, but for the environment it was codded for -
** two threads is more than enough.
** First thread accepts connections, second communicates with clients.
**
**
** This server should be launched on one of the nodes. Other clients should
** know server's ip. Because of the asynchronous design, it produces
** very little overhead.
** When received SIGINT - all connection should be closed and program terminated.
** Check PYSSC git for a client version.
** Although this server was tested with many threads and for a long time,
** it may still have some error or space for improvement. I would be glad to hear
** any response.
**
** Code sucessfully passedd PVS-Studio and Valgrind check.
 */
//============================================================================
// Name        : file_scheduler.cpp
// Author      : Ivan Syzonenko
// Version     :
// Copyright   : MIT
// Description : This server is used to synchronize pssc workers that ran at different nodes
//============================================================================
#include <fcntl.h>
#include <netinet/in.h>
#include <pthread.h>
#include <stddef.h>
#include <string.h>
#include <sys/epoll.h>
#include <sys/socket.h>
#include <unistd.h>
#include <cerrno>
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
#include <sstream>
#include <queue>
#include <deque>
#include <fstream>
#include <ctime>
#include <iomanip>
#include <arpa/inet.h>
#include <csignal>

using std::string;
using std::cerr;
using std::cout;
using std::cin;
using std::endl;
using std::vector;
using std::queue;
using std::deque;
using std::put_time;

// enables log files
//#define DEBUG
// One server generates around 20-25 events.
// For 4 nodes I expect 100 events for cluster
#define MAXEVENTS 500

//just wrapper for better understanding.
struct fd_struct
{
    int fd; ///just wrapper for better understanding.
```

```
};

struct thread_data
{
    queue <fd_struct> file_descriptors;
};

struct client_buffer
{
    int pid;
    int fd;
    string operation;
    string target;
    string answer;
};

struct read_add
{
    int fd;
    string buf;
};

size_t parse_buffer(string str, deque <client_buffer> *client_buf, int fd);
int secure_send(client_buffer* client_buf);
void check_client_errors(deque <client_buffer> *processed_client_buf, deque
 <client_buffer> *client_buf, const ssize_t fd);
void *read_and_respond(void * threadarg);
int accept_connections(uint16_t port, queue <fd_struct> *clients);

pthread_mutex_t lock;
bool time_to_exit = false;
int exit_code = 0;

void signalHandler( int signum )
{
   cerr << "Interrupt signal (" << signum << ") received.\n";
   cerr << "Programm has 30 seconds to finish or it will be forced to exit.\n";
   exit_code = signum;
   time_to_exit = true;
   sleep(30);
   cerr << "Looks like program does not respond. Killing.\n";
   exit(signum);
}

/*!
Parses input buffer and stores parsed messages in queue
It may parse more than one message(stored in str) and if last massage is incomplete - returns how many
 characters to save in external buffer for future processing.
\param[in] str Input buffer with data recieved in socket fd.
\param[in] client_buf Structure that keeps all parsed messages.
\param[in] fd file descriptor associated with passed buffer data.
\return how many characters to save in external buffer for future processing
*/
size_t parse_buffer(string str, deque <client_buffer> *client_buf, int fd)
{
    bool done = false;
    bool error = false;
    client_buffer temp;
    vector <string> info_block;
    string current_str;
    size_t found = 0;
    string token;

    while(str.length() > 0 && !done && !error)
    {
        found = str.find_first_of("#");
        if(found == std::string::npos)
        {
            error = true;
            continue;
        }

        unsigned fut_len = stoi(str.substr(0, found));
        if(str.length() < found + 1 + fut_len)
            break;

        current_str = str.substr(found+1, fut_len);
        str = str.substr(found + 1 + fut_len);

        std::istringstream iss(current_str);
        info_block.clear();
        while (getline(iss, token, '#'))
            info_block.push_back(token);

        current_str.clear();

        temp = {};
        if(info_block.size() > 0 )
            temp.pid = stoi(info_block[0]);

        if(info_block.size() > 1 )
            temp.operation = info_block[1];

        if(info_block.size() > 2 )
            temp.target = info_block[2];

        temp.fd = fd;

        if(temp.operation == "DONE")
            client_buf->push_front(temp);
        else
            client_buf->push_back(temp);

        if(str.length() < 3)
            done = true;
    }

    return str.length();
}

int secure_send(client_buffer* client_buf)
{
    for(size_t as = 0; as < client_buf->answer.length();)
    {
        auto sent = send(client_buf->fd, client_buf->answer.substr(as).c_str(), client_buf->
answer.substr(as).length(), 0);
        if(sent < 0)
        {
            cerr << "Error on socket " << client_buf->fd << endl;
            return -1;
        }
        as += sent;
    }
    return 0;
}

void check_client_errors(deque <client_buffer> *processed_client_buf, deque
 <client_buffer> *client_buf, const ssize_t fd)
{
    string error_target = "";

    for(auto iter = client_buf->begin(); iter != client_buf->end(); ++iter)
    {
        if(iter->fd == fd)
            return;
    }

    for(auto iter = processed_client_buf->begin(); iter != processed_client_buf->end(); ++iter)
    {
        if(iter->fd == fd )
        {
            error_target = iter->target;
            cerr << "Broken client removing: " << iter->fd << " " << iter->operation << " " << iter->target
 << endl;
            processed_client_buf->erase(iter);
            break;
        }
    }
    if(error_target.length() > 0)
    {
        for(auto iter = processed_client_buf->begin(); iter != processed_client_buf->end(); ++iter)
        {
            if(iter->target == error_target && iter->answer == "WAIT")
            {
                iter->answer = "WRIT";
                cerr << "PID " << iter->pid << " advised to WRIT";
                if(secure_send(&*iter) != 0)
                    cerr << "ERROR in secure send";
                break;
            }
        }
    }
}

/*!
Registers new sockets in epoll function(waits on data in async mode). Reads data from sockets in async
 mode. Calls parse function and makes decision according to the processed requests.
\param[in] threadarg Structure that contains address of queue with file descriptors.
*/
void *read_and_respond(void * threadarg)
{
    auto my_data = (thread_data *) threadarg;
    queue <fd_struct> *fds = &my_data->file_descriptors;
    struct epoll_event event;
    struct epoll_event *events;
    auto efd = epoll_create1 (0);
    vector <int> fd_to_remove;
//  vector <size_t> local_fd;
    deque <client_buffer> client_buf;
    deque <client_buffer> processed_client_buf;
    vector <read_add> buff_add;
    read_add ra;
    events = (epoll_event*)calloc (MAXEVENTS, sizeof event);
    event.events = EPOLLIN | EPOLLET;
    size_t fd_event_counter = 0;
    int n;
    std::ofstream log_processing;
    log_processing.open("processing.log", std::ios::out | std::ios::app);
    #ifdef DEBUG
        auto t = time(nullptr);
        auto tm = *localtime(&t);
        log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Thread created\n";
    #endif

    while(!time_to_exit)
    {
        #ifdef DEBUG
            if(fd_to_remove.size() > 0)
            {
                t = time(nullptr);
                tm = *localtime(&t);
                log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Descriptors to clean: " <<
fd_to_remove.size() << "\n";
            }
        #endif
        for(unsigned j=0; j < fd_to_remove.size(); ++j)
        {
            for(size_t k = 0; k < buff_add.size(); ++k)
            {
                if(buff_add[k].fd == fd_to_remove[j])
                {
                    #ifdef DEBUG
                        t = time(nullptr);
                        tm = *localtime(&t);
                        log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Removing " <<
fd_to_remove[j] << endl;
                    #endif
                    buff_add.erase(buff_add.begin() + k);
                    break;
                }
            }

            for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end();)
            {
                if(iter->fd == fd_to_remove[j])
                    iter = processed_client_buf.erase(iter);
                else
                    iter++;
            }

        }

        fd_event_counter -= fd_to_remove.size();
        fd_to_remove.clear();

        #ifdef DEBUG
            if(!fds->empty())
            {
                t = time(nullptr);
                tm = *localtime(&t);
                log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Processing new connections\n";
                if(!fds->empty())
                    log_processing.seekp(-1, std::ios_base::cur);
            }
        #endif

        while(!fds->empty())
        {
            pthread_mutex_lock(&lock);
            event.data.fd = fds->front().fd;
            fds->pop();
            pthread_mutex_unlock(&lock);
            ++fd_event_counter;
            #ifdef DEBUG
                log_processing << ".";
            #endif

            if( epoll_ctl(efd, EPOLL_CTL_ADD, event.data.fd, &event) == -1)
            {
                perror ("epoll_ctl");
                cerr << "EPOLL ERROR\n";
                time_to_exit = true;
                #ifdef DEBUG
                    log_processing.seekp(0, std::ios_base::end);
                    t = time(nullptr);
                    tm = *localtime(&t);
                    log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "EPOLL ERROR\n";
                    log_processing.close();
                #endif
                pthread_exit(NULL);
            }
        }

        #ifdef DEBUG
            log_processing.seekp(0, std::ios_base::end);
        #endif

        if(fd_event_counter > 0)
        {
            n = epoll_wait(efd, events, MAXEVENTS, 1000);
            for(int i = 0; i < n; ++i)
            {
                if (( &events[i] != NULL) && ((events[i].events & EPOLLERR) || (events[i].events & EPOLLHUP) ||  (!(events[i].events & EPOLLIN))))
                {
                    cerr << "epoll error\n";
                    close (events[i].data.fd);
                    fd_to_remove.push_back(events[i].data.fd);
                    continue;
                }
                else
                {
                    int done = 0;
                    ra.buf = "";

                    for(unsigned k = 0; k < buff_add.size(); ++k)
                    {
                        if(buff_add[k].fd == events[i].data.fd)
                        {
                            ra.buf = buff_add[k].buf;
                            break;
                        }
                    }
                    #ifdef DEBUG
                        log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Reading from socket " << events[i].data.fd << endl;
                    #endif
                    while (1)
                    {
                        ssize_t count;
                        char buf[512];
                        memset(buf,0, sizeof buf);
                        count = recv(events[i].data.fd, buf, sizeof buf, 0);
                        #ifdef DEBUG
                                t = time(nullptr);
                                tm = *localtime(&t);
                                log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Recieved: " << string(buf) << endl;
                        #endif
                        if (count == -1)
                        { // If errno == EAGAIN, that means we have read all data. So go back to the main loop.
                            if (errno != EAGAIN)
                            {
                                perror ("read");
                                cerr << "Count error";
                                #ifdef DEBUG
                                t = time(nullptr);
                                tm = *localtime(&t);
                                log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "ERROR: Count error in epoll" << endl;
                                #endif
                                done = 1;
                            }
                            break;
                        }
                        else if (count == 0)
                        { // End of file. The remote has closed the connection.
                            done = 1;
                            break;
                        }
                        ra.buf += string(buf);
                        #ifdef DEBUG
                            t = time(nullptr);
                            tm = *localtime(&t);
                            log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Parsing message of length " << ra.buf.length() << endl;
                            log_processing << "Message: " << string(ra.buf) << endl;
                        #endif
                        size_t char_left = parse_buffer(ra.buf, &client_buf, events[i].data.fd);
                        #ifdef DEBUG
                        t = time(nullptr);
                        tm = *localtime(&t);
                        if(client_buf.size() > 0)
                        {
                            for(auto iter = client_buf.begin(); iter != client_buf.end(); ++iter)
                            {
                                log_processing << "PID " << iter->pid << " on " << iter->fd << " requested " << iter->operation << " " << iter->target << endl;
                            }
                        }
                        else
                            log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "WARNING: Client buffer is empty\n";

                        #endif
                        if(char_left > 0)
                        {
                            #ifdef DEBUG
                                t = time(nullptr);
                                tm = *localtime(&t);
                                log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Not all information was received. Missing " << char_left << " chars\n";
                            #endif

                            ra.fd = events[i].data.fd;
                            unsigned k = 0;
                            for(; k < buff_add.size(); ++k)
                            {
                                if(buff_add[k].fd == events[i].data.fd)
                                {
                                    buff_add[k].buf += ra.buf;
                                    break;
                                }
                            }
                            if(k == buff_add.size())
                                buff_add.push_back(ra);
                        }
                        else
                        {
                            #ifdef DEBUG
                                t = time(nullptr);
                                tm = *localtime(&t);
                                log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Success in reading full message\n";
                            #endif
                            for(unsigned k = 0; k < buff_add.size(); ++k)
                            {
                                if(buff_add[k].fd == events[i].data.fd)
                                {
                                    buff_add.erase(buff_add.begin() + k);
                                    break;
                                }
                            }
                        }
                    }

                    if (done)
                    {
                        #ifdef DEBUG
                        t = time(nullptr);
                        tm = *localtime(&t);
                        log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Closing connection on descriptor " << events[i].data.fd << endl;
                        #endif

                        check_client_errors(&processed_client_buf, &client_buf, events[i].data.fd);

                        close (events[i].data.fd); // Closing the descriptor will make epoll remove it from the set of descriptors which are monitored.
                        fd_to_remove.push_back(events[i].data.fd);
                    }
                }
            }

            #ifdef DEBUG
            if(!client_buf.empty())
            {
                t = time(nullptr);
                tm = *localtime(&t);
                log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Begin processing requests.\n";
            }
            #endif
```

```cpp

            #ifdef DEBUG
            log_processing << "processed_client_buf before processing new requests \n************\n";
            for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end(); ++iter)
            {
                log_processing << "PID " << iter->pid << " from socket " << iter->fd << " requested " << iter->operation << " " << iter->target << " advised " << iter->answer << endl;
            }
            log_processing << "************\n end\n";
            #endif

            while(!client_buf.empty())
            {
                if(client_buf.front().operation == "READ")
                {
                    #ifdef DEBUG
                        t = time(nullptr);
                        tm = *localtime(&t);
                        log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "PID: " << client_buf.front().pid << " from socket " << client_buf.front().fd << " wants to read " << client_buf.front().target << endl;
                    #endif
                    client_buf.front().answer = "READ";

                    for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end(); ++iter)
                    {
                        if(iter->target == client_buf.front().target )
                        {
                            if(iter->answer == "WRIT" || iter->answer == "WAIT")
                            {
                                client_buf.front().answer = "WAIT";
                                break;
                            }
                            else if(iter->answer == "READ")
                            {
                                client_buf.front().answer = "READ";
                                break;
                            }
                        }
                    }

                    if(secure_send(&client_buf.front()) == 0)
                        processed_client_buf.push_back(client_buf.front());
                    else
                        cerr << "ERROR in secure send";

                    #ifdef DEBUG
                        t = time(nullptr);
                        tm = *localtime(&t);
                        log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "RESPONSE to PID: " << client_buf.front().pid << " from socket " << client_buf.front().fd << ": " << client_buf.front().answer << endl;
                    #endif
                }
                else if(client_buf.front().operation == "WRIT")
                {
                    #ifdef DEBUG
                        t = time(nullptr);
                        tm = *localtime(&t);
                        log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "PID: " << client_buf.front().pid << " from socket " << client_buf.front().fd << " wants to write " << client_buf.front().target << endl;
                    #endif
                    client_buf.front().answer = "WRIT";
                    for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end(); ++iter)
                    {
                        if(iter->target == client_buf.front().target )
                        {
                            if(iter->answer == "WRIT" || iter->answer == "WAIT")
                            {
                                client_buf.front().answer = "WAIT";
                                break;
                            }
                            else if(iter->answer == "READ")
                            {
                                client_buf.front().answer = "READ";
                                break;
                            }
                        }
                    }

                    if(secure_send(&client_buf.front()) == 0)
                        processed_client_buf.push_back(client_buf.front());
                    else
                        cerr << "ERROR in secure send";

                    #ifdef DEBUG
                        t = time(nullptr);
                        tm = *localtime(&t);
                        log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "RESPONSE to PID: " << client_buf.front().pid << " from socket " << client_buf.front().fd << ": " << client_buf.front().answer << endl;
                    #endif
                }
                else if(client_buf.front().operation == "DONE")
                {
                    #ifdef DEBUG
                        t = time(nullptr);
                        tm = *localtime(&t);
                        log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "PID: " << client_buf.front().pid << " from socket " << client_buf.front().fd << " finished working with " << client_buf.front().target << endl;
                    #endif

                    for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end();)
                    {
                        if(iter->target == client_buf.front().target )
                        {
                            if(iter->pid == client_buf.front().pid)
                            {
                                #ifdef DEBUG
                                    t = time(nullptr);
                                    tm = *localtime(&t);
                                    log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "PID: " << iter->pid << " from socket " << iter->fd << ": is DONE - SELF-DESTRUCTION" << endl;
                                #endif
                                iter = processed_client_buf.erase(iter);
                                continue;
                            }
                            else if(iter->answer == "WAIT")
                            {
                                iter->answer = "READ";
                                if(secure_send(&*iter) != 0)
                                    cerr << "ERROR in secure send";

                                #ifdef DEBUG
                                    t = time(nullptr);
                                    tm = *localtime(&t);
                                    log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "RESPONSE to PID: " << iter->pid << " from socket " << iter->fd << ": " << iter->answer << endl;
                                #endif
                            }
                        }

                        ++iter;
                    }
                }
                else
                    cerr << client_buf.front().operation << endl;

                client_buf.pop_front();
            }

            #ifdef DEBUG
            log_processing << "processed_client_buf after processing new requests \n************\n";
            for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end(); ++iter)
            {
                log_processing << "PID " << iter->pid << " from socket " << iter->fd << " requested " << iter->operation << " " << iter->target << " advised " << iter->answer << endl;
            }
            log_processing << "************\n end\n";
            #endif
        }
        else
            sleep(1);
    }

    for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end();)
    {
        iter->answer = "EXIT";
//      cerr <<
        secure_send(&*iter);
        close(iter->fd);
    }
    processed_client_buf.clear();

    #ifdef DEBUG
        log_processing.seekp(0, std::ios_base::end);
        t = time(nullptr);
        tm = *localtime(&t);
        log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Time to exit\n";
        log_processing.close();
    #endif

    pthread_exit(NULL);
}

/*!
Initializes socket on port 'port' and waits for connections. On incomming connection sets associated socket
 to async mode
and stores in fd_struct structure which is shared with processing thread.
\param[in] port Port used to create listening socket.
\param[in] clients queue that stores file descriptors(sockets) accepted on port 'port'.
\returns status code
*/
int accept_connections(uint16_t port, queue <fd_struct> *clients)
{
    std::string rcv;
    int listen_fd, comm_fd;
    struct sockaddr_in servaddr;
    std::ofstream log_main;
    log_main.open("incoming.log", std::ios::out | std::ios::app);
    #ifdef DEBUG
        auto t = time(nullptr);
        auto tm = *localtime(&t);
        log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Thread created\n";
    #endif

    listen_fd = socket(AF_INET, SOCK_STREAM, 0);

    if (listen_fd == -1)
    {
        cout << "Can't create file descriptor." << endl;
        exit_code = 1;
        time_to_exit = true;
        sleep(10);
        exit(1);
    }

    memset( &servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htons(INADDR_ANY);
    servaddr.sin_port = htons(port);
    #ifdef DEBUG
        t = time(nullptr);
        tm = *localtime(&t);
        log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Attempting to listen on " << port << " port\n
";
    #endif
    int my_timer = 20;

    while(my_timer > 0)
    {
        if(bind(listen_fd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0)
            sleep(10);
        else
            break;
        my_timer--;
    }

    if(my_timer == 0)
    {
        cout << "Binding to socket error." << endl;
        exit_code = 1;
        time_to_exit = true;
        sleep(10);
        exit(2);
    }
    #ifdef DEBUG
        t = time(nullptr);
        tm = *localtime(&t);
        log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Successful bind to the socket\n";
    #endif

    fd_struct temp;

    while(!time_to_exit)
    {
        listen(listen_fd, 60);
        #ifdef DEBUG
            t = time(nullptr);
            tm = *localtime(&t);
            log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Waiting for incoming connections.\n";
            log_main.flush();
        #endif
        sockaddr_in clientAddr;
        socklen_t sin_size=sizeof(struct sockaddr_in);
        comm_fd = accept(listen_fd, (struct sockaddr*)&clientAddr, &sin_size);

        if(comm_fd == -1)
        {
            cout << "Connection acceptance error." << endl;
            //      exit(3);
        }

        #ifdef DEBUG
            t = time(nullptr);
            tm = *localtime(&t);
            char loc_addr[INET_ADDRSTRLEN+1];
            inet_ntop(AF_INET, &(clientAddr.sin_addr), loc_addr, INET_ADDRSTRLEN);
            log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Incoming connection on descriptor " << comm_fd << " from " << loc_addr << ":" << clientAddr.sin_port <<"\n";
        #endif
        //      make nonblocking
        auto flags = fcntl (comm_fd, F_GETFL, 0);
        if (flags < 0)
        {
            perror ("fcntl");
            return -1;
        }

        flags |= O_NONBLOCK;
        auto s = fcntl (comm_fd, F_SETFL, flags);

        if(s < 0)
        {
            perror ("fcntl");
            time_to_exit = true;
            log_main.close();
            return -1;
        }
        #ifdef DEBUG
            t = time(nullptr);
            tm = *localtime(&t);
            log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Switched " << comm_fd <<" to nonblocking mode.\n";
        #endif

        temp.fd = comm_fd;

        pthread_mutex_lock(&lock);
            clients->push(temp);
        pthread_mutex_unlock(&lock);
        #ifdef DEBUG
            t = time(nullptr);
            tm = *localtime(&t);
            log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Successfully pushed into queue for further processing.\n";
        #endif
    }

    #ifdef DEBUG
    log_main.close();
    #endif

    return 0;
}

/*!
Nothing fancy. Creates a thread and launches connection accepting function
\returns status code to OS
\param clients data structure to store fd
*/
int main()
{
    queue <fd_struct> clients;
    pthread_t threads[1];

    signal(SIGINT, signalHandler);

    if (pthread_mutex_init(&lock, NULL) != 0)
    {
        printf("\n mutex init failed\n");
        return 1;
    }

    int rc = pthread_create(&threads[0], NULL, read_and_respond, (void *)&clients);
    if (rc)
    {
        cout << "Error:unable to create thread," << rc << endl;
        exit_code = -1;
        time_to_exit = true;
        sleep(10);
        return exit_code;

    }

    accept_connections(1987, &clients);

    pthread_join(rc, NULL);
    pthread_mutex_destroy(&lock);

    return exit_code;
}
```

5.3 README.md File Reference

5.4 README.md

```
# PYSSC scheduler.

This server is used to synchronize pssc workers that run at different nodes.

Main problem is that each of them have to know whether to generate 'reuse' file
read it or wait while it is being generated by other worker(node).

This server recieves data about requested file and respond with 'suggestion':
READ, WRIT, WAIT.

In short. If file does not exist and is not being generated - generate it (WRIT).
If file exists - read it (READ). Read operation does not change the data, so can be
done in parallel, anyway file will be cached into RAM.
If file is being generated - WAIT for a next READ message.

Communication is done by epoll. It is possible to add any reasonable number
of threads if needed, but for the environment it was codded for -
two threads is more than enough.
First thread accepts connections, second communicates with clients.


This server should be launched on one of the nodes. Other clients should
know server's ip. Because of the asynchronous design, it produces
very little overhead.
When received SIGINT - all connection should be closed and program terminated.
Check PYSSC git for a client version.
Although this server was tested with many threads and for a long time,
it may still have some error or space for improvement. I would be glad to hear
any response.

Code sucessfully passedd PVS-Studio and Valgrind check.
```

Index