Saurion

Generated by Doxygen 1.9.4

Todo List	1
Module Index	3
2.1 Modules	3
Class Index	5
3.1 Class List	5
Eila Inday	7
File Index 4.1 File List	-
4.1 File List	/
Module Documentation	9
5.1 LowSaurion	9
5.1.1 Detailed Description	10
5.1.2 Macro Definition Documentation	12
5.1.2.1 _POSIX_C_SOURCE	12
5.1.2.2 PACKING_SZ	12
5.1.3 Function Documentation	12
5.1.3.1 allocate_iovec()	13
5.1.3.2 free_request()	13
5.1.3.3 initialize_iovec()	13
5.1.3.4 read_chunk()	15
5.1.3.5 saurion_create()	16
5.1.3.6 saurion_destroy()	18
5.1.3.7 saurion_send()	19
5.1.3.8 saurion_set_socket()	19
5.1.3.9 saurion_start()	20
5.1.3.10 saurion_stop()	21
5.1.3.11 set_request()	21
5.2 ThreadPool	
5.2.1 Detailed Description	23
5.2.2 Function Documentation	23
5.2.2.1 threadpool_add()	23
5.2.2.2 threadpool_create()	24
5.2.2.3 threadpool_create_default()	25
5.2.2.4 threadpool_destroy()	
5.2.2.5 threadpool_empty()	26
5.2.2.6 threadpool_init()	
5.2.2.7 threadpool_stop()	
5.2.2.8 threadpool_wait_empty()	
Class Documentation	29
6.1 chunk_params Struct Reference	
6.1.1 Detailed Description	29

6.1.2 Member Data Documentation	29
6.1.2.1 cont_rem	29
6.1.2.2 cont_sz	30
6.1.2.3 curr_iov	30
6.1.2.4 curr_iov_off	30
6.1.2.5 dest	30
6.1.2.6 dest_off	30
6.1.2.7 dest_ptr	30
6.1.2.8 len	31
6.1.2.9 max_iov_cont	31
6.1.2.10 req	31
6.2 Node Struct Reference	31
6.2.1 Detailed Description	31
6.2.2 Member Data Documentation	31
6.2.2.1 children	32
6.2.2.2 next	32
6.2.2.3 ptr	32
6.2.2.4 size	32
6.3 request Struct Reference	32
6.3.1 Detailed Description	33
6.3.2 Member Data Documentation	33
6.3.2.1 client_socket	33
6.3.2.2 event_type	33
6.3.2.3 iov	33
6.3.2.4 iovec_count	33
6.3.2.5 next_iov	33
6.3.2.6 next_offset	34
6.3.2.7 prev	34
6.3.2.8 prev_remain	34
6.3.2.9 prev_size	34
6.4 saurion Struct Reference	34
6.4.1 Detailed Description	35
6.4.2 Member Data Documentation	35
6.4.2.1 efds	35
6.4.2.2 list	35
6.4.2.3 m_rings	36
6.4.2.4 n_threads	36
6.4.2.5 next	36
6.4.2.6 pool	36
6.4.2.7 rings	36
6.4.2.8 ss	37
6.4.2.9 status	37

6.4.2.10 status_c	37
6.4.2.11 status_m	37
6.5 Saurion Class Reference	37
6.5.1 Detailed Description	38
6.5.2 Member Typedef Documentation	38
6.5.2.1 ClosedCb	38
6.5.2.2 ConnectedCb	39
6.5.2.3 ErrorCb	39
6.5.2.4 ReadedCb	39
6.5.2.5 WroteCb	39
6.5.3 Constructor & Destructor Documentation	39
6.5.3.1 Saurion() [1/3]	39
6.5.3.2 ~Saurion()	40
6.5.3.3 Saurion() [2/3]	40
6.5.3.4 Saurion() [3/3]	40
6.5.4 Member Function Documentation	40
6.5.4.1 init()	40
6.5.4.2 on_closed()	40
6.5.4.3 on_connected()	41
6.5.4.4 on_error()	41
6.5.4.5 on_readed()	41
6.5.4.6 on_wrote()	41
6.5.4.7 operator=() [1/2]	42
6.5.4.8 operator=() [2/2]	42
6.5.4.9 send()	42
6.5.4.10 stop()	42
6.5.5 Member Data Documentation	42
6.5.5.1 s	42
6.6 saurion::saurion_callbacks Struct Reference	43
6.6.1 Detailed Description	43
6.6.2 Member Data Documentation	43
6.6.2.1 on_closed	43
6.6.2.2 on_closed_arg	44
6.6.2.3 on_connected	44
6.6.2.4 on_connected_arg	44
6.6.2.5 on_error	44
6.6.2.6 on_error_arg	45
6.6.2.7 on_readed	45
6.6.2.8 on_readed_arg	45
6.6.2.9 on_wrote	45
6.6.2.10 on_wrote_arg	46
6.7 saurion, callbacks Struct Reference	46

6.7.1 Detailed Description	. 47
6.7.2 Member Data Documentation	. 47
6.7.2.1 on_closed	. 47
6.7.2.2 on_closed_arg	. 47
6.7.2.3 on_connected	. 47
6.7.2.4 on_connected_arg	. 48
6.7.2.5 on_error	. 48
6.7.2.6 on_error_arg	. 48
6.7.2.7 on_readed	. 48
6.7.2.8 on_readed_arg	. 49
6.7.2.9 on_wrote	. 49
6.7.2.10 on_wrote_arg	. 49
6.8 saurion_wrapper Struct Reference	. 50
6.8.1 Detailed Description	. 50
6.8.2 Member Data Documentation	. 50
6.8.2.1 s	. 50
6.8.2.2 sel	. 50
6.9 task Struct Reference	
6.9.1 Detailed Description	. 51
6.9.2 Member Data Documentation	. 51
6.9.2.1 argument	. 51
6.9.2.2 function	. 51
6.9.2.3 next	. 51
6.10 threadpool Struct Reference	. 51
6.10.1 Detailed Description	
6.10.2 Member Data Documentation	
6.10.2.1 empty_cond	. 52
6.10.2.2 num_threads	
6.10.2.3 queue_cond	. 52
6.10.2.4 queue_lock	
6.10.2.5 started	
6.10.2.6 stop	
6.10.2.7 task_queue_head	
6.10.2.8 task_queue_tail	
6.10.2.9 threads	. 53
7 File Documentation	55
7.1 /w/saurion/saurion/include/linked_list.h File Reference	. 55
7.1.1 Function Documentation	
7.1.1.1 list_delete_node()	
7.1.1.2 list_free()	
7.1.1.3 list_insert()	

7.2 linked_list.h	7
7.3 /w/saurion/saurion/include/low_saurion.h File Reference	7
7.3.1 Variable Documentation	8
7.3.1.1 efds	9
7.3.1.2 list	9
7.3.1.3 m_rings	9
7.3.1.4 n_threads	9
7.3.1.5 next	9
7.3.1.6 on_closed	9
7.3.1.7 on_closed_arg	0
7.3.1.8 on_connected	0
7.3.1.9 on_connected_arg	0
7.3.1.10 on_error	1
7.3.1.11 on_error_arg	1
7.3.1.12 on_readed	1
7.3.1.13 on_readed_arg	2
7.3.1.14 on_wrote	2
7.3.1.15 on_wrote_arg	2
7.3.1.16 pool	2
7.3.1.17 rings	3
7.3.1.18 ss	3
7.3.1.19 status	3
7.3.1.20 status_c	3
7.3.1.21 status_m	3
7.4 low_saurion.h	4
7.5 /w/saurion/saurion/include/low_saurion_secret.h File Reference	4
7.6 low_saurion_secret.h	5
7.7 /w/saurion/saurion/include/saurion.hpp File Reference	5
7.8 saurion.hpp	6
7.9 /w/saurion/saurion/include/threadpool.h File Reference	6
7.10 threadpool.h	7
7.11 /w/saurion/saurion/src/linked_list.c File Reference	7
7.11.1 Function Documentation	8
7.11.1.1 create_node()	8
7.11.1.2 free_node()	8
7.11.1.3 list_delete_node()	9
7.11.1.4 list_free()	9
7.11.1.5 list_insert()	0
7.11.2 Variable Documentation	0
7.11.2.1 list_mutex	0
7.12 linked_list.c	0
7.13 /w/saurion/src/low_saurion.c File Reference	2

7.13.1 Macro Definition Documentation	74
7.13.1.1 EV_ACC	74
7.13.1.2 EV_ERR	74
7.13.1.3 EV_REA	74
7.13.1.4 EV_WAI	75
7.13.1.5 EV_WRI	75
7.13.1.6 MAX	75
7.13.1.7 MIN	75
7.13.2 Function Documentation	75
7.13.2.1 add_accept()	76
7.13.2.2 add_efd()	76
7.13.2.3 add_fd()	76
7.13.2.4 add_read()	77
7.13.2.5 add_read_continue()	77
7.13.2.6 add_write()	78
7.13.2.7 calculate_max_iov_content()	79
7.13.2.8 copy_data()	79
7.13.2.9 handle_accept()	80
7.13.2.10 handle_close()	80
7.13.2.11 handle_error()	80
7.13.2.12 handle_event_read()	81
7.13.2.13 handle_new_message()	81
7.13.2.14 handle_partial_message()	82
7.13.2.15 handle_previous_message()	82
7.13.2.16 handle_read()	83
7.13.2.17 handle_write()	83
7.13.2.18 htonll()	83
7.13.2.19 next()	84
7.13.2.20 ntohll()	84
7.13.2.21 prepare_destination()	84
7.13.2.22 read_chunk_free()	85
7.13.2.23 saurion_worker_master()	85
7.13.2.24 saurion_worker_master_loop_it()	86
7.13.2.25 saurion_worker_slave()	86
7.13.2.26 saurion_worker_slave_loop_it()	87
7.13.2.27 validate_and_update()	88
7.13.3 Variable Documentation	88
7.13.3.1 TIMEOUT_RETRY_SPEC	88
7.14 low_saurion.c	88
7.15 /w/saurion/saurion/src/main.c File Reference	03
7.16 main.c	03
7.17 /w/saurion/saurion/src/saurion.cpp File Reference	03

7.19.1.1 FALSE 109 7.19.1.2 TRUE 109 7.19.2 Function Documentation 109	5
7.19.1.1 FALSE	5
7.19.1 Macro Definition Documentation	5
w/saurion/saurion/src/threadpool.c File Reference	5
saurion.cpp	4
	saurion.cpp

Chapter 1

Todo List

Member read_chunk (void **dest, size_t *len, struct request *const req)
add message contraint
validar msg_size, crear maximos
validar offsets

2 Todo List

Chapter 2

Module Index

2.1 Modules

Here is a list of all modules:

LowSaurion	 				 					 	 														9
ThreadPool	 		 						 							 								2	3

4 Module Index

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

nunk_params	29
ode	
quest	32
Main structure for managing io_uring and socket events	34
aurion	37
urion::saurion_callbacks	
Structure containing callback functions to handle socket events	13
urion_callbacks	
Structure containing callback functions to handle socket events	16
urion_wrapper	50
sk	50
readpool	51

6 Class Index

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

/_w/saurion/saurion/include/linked_list.h	55
/_w/saurion/saurion/include/low_saurion.h	57
/_w/saurion/saurion/include/low_saurion_secret.h	64
/_w/saurion/saurion/include/saurion.hpp	65
/_w/saurion/saurion/include/threadpool.h	66
/_w/saurion/saurion/src/linked_list.c	67
/_w/saurion/saurion/src/low_saurion.c	72
/_w/saurion/saurion/src/main.c	103
/_w/saurion/saurion/src/saurion.cpp	103
/ w/saurion/saurion/src/threadpool.c	105

8 File Index

Chapter 5

Module Documentation

5.1 LowSaurion

The saurion class is designed to efficiently handle asynchronous input/output events on Linux systems using the io_uring API. Its main purpose is to manage network operations such as socket connections, reads, writes, and closures by leveraging an event-driven model that enhances performance and scalability in highly concurrent applications.

Classes

· struct saurion

Main structure for managing io_uring and socket events.

Macros

- #define POSIX C SOURCE 200809L
- #define PACKING_SZ 32

Defines the memory alignment size for structures in the saurion class.

Functions

• int saurion_set_socket (int p)

Creates a socket.

struct saurion * saurion_create (uint32_t n_threads)

Creates an instance of the saurion structure.

int saurion_start (struct saurion *s)

Starts event processing in the saurion structure.

void saurion_stop (const struct saurion *s)

Stops event processing in the saurion structure.

void saurion_destroy (struct saurion *s)

Destroys the saurion structure and frees all associated resources.

void saurion_send (struct saurion *s, const int fd, const char *const msg)

Sends a message through a socket using io_uring.

- int allocate_iovec (struct iovec *iov, size_t amount, size_t pos, size_t size, void **chd_ptr)
- int initialize_iovec (struct iovec *iov, size_t amount, size_t pos, const void *msg, size_t size, uint8_t h)

Initializes a specified iovec structure with a message fragment.

• int set_request (struct request **r, struct Node **I, size_t s, const void *m, uint8_t h)

Sets up a request and allocates iovec structures for data handling in liburing.

int read_chunk (void **dest, size_t *len, struct request *const req)

Reads a message chunk from the request's iovec buffers, handling messages that may span multiple iovec entries.

void free_request (struct request *req, void **children_ptr, size_t amount)

5.1.1 Detailed Description

The saurion class is designed to efficiently handle asynchronous input/output events on Linux systems using the io_uring API. Its main purpose is to manage network operations such as socket connections, reads, writes, and closures by leveraging an event-driven model that enhances performance and scalability in highly concurrent applications.

This function allocates memory for each struct iovec

The main structure, saurion, encapsulates io_uring rings and facilitates synchronization between multiple threads through the use of mutexes and a thread pool that distributes operations in parallel. This allows efficient handling of I/O operations across several sockets simultaneously, without blocking threads during operations.

The messages are composed of three main parts:

- A header, which is an unsigned 64-bit number representing the length of the message body.
- · A body, which contains the actual message data.
- · A footer, which consists of 8 bits set to 0.

For example, for a message with 9000 bytes of content, the header would contain the number 9000, the body would consist of those 9000 bytes, and the footer would be 1 byte set to 0.

When these messages are sent to the kernel, they are divided into chunks using iovec. Each chunk can hold a maximum of 8192 bytes and contains two fields:

- iov_base, which is an array where the chunk of the message is stored.
- iov_len, the number of bytes used in the iov_base array.

For the message with 9000 bytes, the iovec division would look like this:

- The first iovec would contain:
 - 8 bytes for the header (the length of the message body, 9000).
 - 8184 bytes of the message body.
 - iov_len would be 8192 bytes in total.
- The second iovec would contain:
 - The remaining 816 bytes of the message body.
 - 1 byte for the footer (set to 0).
 - iov_len would be 817 bytes in total.

The structure of the message is as follows:

+			-+-		+-	+
	Header			Body		Footer
	(64 bits:	9000)		(Message Data)		(1 byte)
+			-+-		+-	+

The structure of the iovec division is:

Each I/O event can be monitored and managed through custom callbacks that handle connection, read, write, close, or error events on the sockets.

Basic usage example:

```
// Create the saurion structure with 4 threads
struct saurion *s = saurion_create(4);
// Start event processing
if (saurion_start(s) != 0) {
    // Handle the error
}
// Send a message through a socket
saurion_send(s, socket_fd, "Hello, World!");
// Stop event processing
saurion_stop(s);
// Destroy the structure and free resources
saurion_destroy(s);
```

In this example, the saurion structure is created with 4 threads to handle the workload. Event processing is started, allowing it to accept connections and manage I/O operations on sockets. After sending a message through a socket, the system can be stopped, and the resources are freed.

Author

Israel

Date

2024

This function allocates memory for each struct iovec. Every struct iovec consists of two member variables:

- iov_base, a void * array that will hold the data. All of them will allocate the same amount of memory (CHUNK_SZ) to avoid memory fragmentation.
- iov_len, an integer representing the size of the data stored in the iovec. The data size is CHUNK_SZ unless it's the last one, in which case it will hold the remaining bytes. In addition to initialization, the function adds the pointers to the allocated memory into a child array to simplify memory deallocation later on.

Parameters

iov	Structure to initialize.
amount	Total number of iovec to initialize.
pos	Current position of the iovec within the total iovec (amount).
size	Total size of the data to be stored in the iovec.
chd_ptr	Array to hold the pointers to the allocated memory.

Return values

ERROR_CODE	if there was an error during memory allocation.
SUCCESS_CODE	if the operation was successful.

Note

The last iovec will allocate only the remaining bytes if the total size is not a multiple of CHUNK SZ.

5.1.2 Macro Definition Documentation

5.1.2.1 POSIX C SOURCE

#define _POSIX_C_SOURCE 200809L

Definition at line 107 of file low_saurion.h.

5.1.2.2 PACKING_SZ

#define PACKING_SZ 32

Defines the memory alignment size for structures in the saurion class.

PACKING_SZ is used to ensure that certain structures, such as saurion_callbacks, are aligned to a specific memory boundary. This can improve memory access performance and ensure compatibility with certain hardware architectures that require specific alignment.

In this case, the value is set to 32 bytes, meaning that structures marked with $__$ attribute $__$ ((aligned(\leftarrow PACKING_SZ))) will be aligned to 32-byte boundaries.

Proper alignment can be particularly important in multithreaded environments or when working with low-level system APIs like io_uring, where unaligned memory accesses may introduce performance penalties.

Adjusting PACKING_SZ may be necessary depending on the hardware platform or specific performance requirements.

Definition at line 139 of file low_saurion.h.

5.1.3 Function Documentation

5.1.3.1 allocate_iovec()

Definition at line 164 of file low saurion.c.

```
00166 {
00167
        if (!iov || !chd_ptr)
00168
        {
00169
            return ERROR_CODE;
00170
00171
        iov->iov_base = malloc (CHUNK_SZ);
00172
        if (!iov->iov_base)
00173
         {
00174
            return ERROR_CODE;
00175
       iov->iov_len = (pos == (amount - 1) ? (size % CHUNK_SZ) : CHUNK_SZ);
if (iov->iov_len == 0)
00176
00177
00178
        {
00179
            iov->iov_len = CHUNK_SZ;
00180
00181
       chd_ptr[pos] = iov->iov_base;
00182
       return SUCCESS_CODE;
00183 }
```

5.1.3.2 free_request()

Definition at line 94 of file low_saurion.c.

```
00095 {
00096
        if (children_ptr)
00097
00098
            free (children_ptr);
00099
           children_ptr = NULL;
00100
00101
       for (size_t i = 0; i < amount; ++i)</pre>
00102
           free (req->iov[i].iov_base);
00103
00104
           req->iov[i].iov_base = NULL;
00105
00106
       free (req);
00107
       req = NULL;
       free (children_ptr);
00108
00109
       children_ptr = NULL;
00110 }
```

5.1.3.3 initialize_iovec()

```
size_t size,
uint8_t h ) [private]
```

Initializes a specified iovec structure with a message fragment.

This function populates the iov_base of the iov_c structure with a portion of the message, depending on the position (pos) in the overall set of iovec structures. The message is divided into chunks, and for the first iov_c , a header containing the size of the message is included. Optionally, padding or adjustments can be applied based on the h flag.

Parameters

iov	Pointer to the iovec structure to initialize.
amount	The total number of iovec structures.
pos	The current position of the iovec within the overall message split.
msg	Pointer to the message to be split across the iovec structures.
size	The total size of the message.
h	A flag (header flag) that indicates whether special handling is needed for the first iovec (adds the message size as a header) or for the last chunk.

Return values

SUCCESS_CODE	on successful initialization of the iovec.
ERROR_CODE	if the iov or its iov_base is null.

Note

For the first iovec (when pos == 0), the message size is copied into the beginning of the iov_base if the header flag (h) is set. Subsequent chunks are filled with message data, and the last chunk may have one byte reduced if h is set.

Attention

The message must be properly aligned and divided, especially when using the header flag to ensure no memory access issues.

Warning

If msg is null, the function will initialize the iov_base with zeros, essentially resetting the buffer.

Definition at line 115 of file low saurion.c.

```
00117 {
        if (!iov || !iov->iov_base)
00118
00119
         {
00120
             return ERROR_CODE;
00121
00122
           (msg)
00123
             size_t len = iov->iov_len;
00124
             char *dest = (char *)iov->iov_base;
char *orig = (char *)msg + pos * CHUNK_SZ;
00125
00126
00127
             size_t cpy_sz = 0;
00128
             if (h)
00129
                 if (pos == 0)
00130
00131
00132
                      uint64_t send_size = htonll (size);
00133
                      memcpy (dest, &send_size, sizeof (uint64_t));
```

```
dest += sizeof (uint64_t);
00135
                    len -= sizeof (uint64_t);
00136
00137
                else
00138
                  {
                    orig -= sizeof (uint64_t);
00139
00140
00141
                if ((pos + 1) == amount)
00142
                     --len;
00143
                    cpy_sz = (len < size ? len : size);
00144
00145
                    dest[cpy_sz] = 0;
00146
00147
00148
            cpy_sz = (len < size ? len : size);
00149
            memcpy (dest, orig, cpy_sz);
            dest += cpy_sz;
size_t rem = CHUNK_SZ - (dest - (char *)iov->iov_base);
00150
00151
00152
            memset (dest, 0, rem);
00153
00154
00155
00156
            memset ((char *)iov->iov_base, 0, CHUNK_SZ);
00157
00158
       return SUCCESS_CODE;
00159 }
```

5.1.3.4 read_chunk()

Reads a message chunk from the request's iovec buffers, handling messages that may span multiple iovec entries.

This function processes data from a struct request, which contains an array of iovec structures representing buffered data. Each message in the buffers starts with a size_t value indicating the size of the message, followed by the message content. The function reads the message size, allocates a buffer for the message content, and copies the data from the iovec buffers into this buffer. It handles messages that span multiple iovec entries and manages incomplete messages by storing partial data within the request structure for subsequent reads.

Parameters

out	dest	Pointer to a variable where the address of the allocated message buffer will be stored. The buffer is allocated by the function and must be freed by the caller.
out	len	Pointer to a size_t variable where the length of the read message will be stored. If a complete message is read, *len is set to the message size. If the message is incomplete, *len is set to 0.
in,o	ut <i>req</i>	Pointer to a struct request containing the lovec buffers and state information. The function updates the request's state to track the current position within the lovecs and any incomplete messages.

Note

The function assumes that each message is prefixed with its size (of type $size_t$), and that messages may span multiple iovec entries. It also assumes that the data in the iovec buffers is valid and properly aligned for reading $size_t$ values.

Warning

The caller is responsible for freeing the allocated message buffer pointed to by *dest when it is no longer needed.

Returns

int Returns SUCCESS_CODE on success, or ERROR_CODE on failure (malformed msg).

Return values

SUCCESS_CODE	No malformed message found.
ERROR_CODE	Malformed message found.

Todo add message contraint

```
validar msg_size, crear maximos
validar offsets
```

Definition at line 688 of file low_saurion.c.

```
00690
        struct chunk_params p;
00691
        p.req = req;
        p.dest = dest;
00692
        p.len = len;
00693
        if (p.req->iovec_count == 0)
00695
00696
            return ERROR_CODE;
         }
00697
00698
00699
       p.max_iov_cont = calculate_max_iov_content (p.req);
00700
       p.cont_sz = 0;
       p.cont_rem = 0;
p.curr_iov = 0;
00701
00702
00703
        p.curr_iov_off = 0;
        p.dest_off = 0;
p.dest_ptr = NULL;
00704
00705
00706
        if (!prepare_destination (&p))
00707
        {
00708
            return ERROR_CODE;
00709
         }
00710
00711
        uint8_t ok = 1UL;
copy_data (&p, &ok);
00712
00713
00714
        if (validate_and_update (&p, ok))
00715
            return SUCCESS_CODE;
00716
00717
00718
       read_chunk_free (&p);
00719
        return ERROR_CODE;
00720 }
```

5.1.3.5 saurion_create()

Creates an instance of the saurion structure.

This function initializes the saurion structure, sets up the eventfd, and configures the io_uring queue, preparing it for use. It also sets up the thread pool and any necessary synchronization mechanisms.

Parameters

n_threads The number of threads to initialize in the thread pool.

Returns

struct saurion* A pointer to the newly created saurion structure, or NULL if an error occurs.

Definition at line 835 of file low_saurion.c.

```
00837
        LOG_INIT (" ");
00838
        struct saurion *p = (struct saurion *)malloc (sizeof (struct saurion));
00839
        if (!p)
00840
          {
00841
            LOG_END (" ");
00842
            return NULL;
00843
        int ret = 0;
00844
00845
        ret = pthread_mutex_init (&p->status_m, NULL);
00846
        if (ret)
00847
          {
            free (p);
LOG_END (" ");
00848
00849
00850
            return NULL;
00851
00852
        ret = pthread cond init (&p->status c, NULL);
00853
        if (ret)
00854
          {
            free (p);
LOG_END (" ");
00855
00856
00857
            return NULL;
00858
00859
        p->m rings
00860
             = (pthread_mutex_t *)malloc (n_threads * sizeof (pthread_mutex_t));
00861
        if (!p->m_rings)
00862
            free (p);
LOG_END (" ");
00863
00864
00865
            return NULL;
00866
00867
        for (uint32_t i = 0; i < n_threads; ++i)</pre>
00868
00869
            pthread_mutex_init (&(p->m_rings[i]), NULL);
00870
00871
        p->ss=0;
        n_threads = (n_threads < 2 ? 2 : n_threads);</pre>
00872
        n_threads = (n_threads > NUM_CORES ? NUM_CORES : n_threads);
00874
        p->n_threads = n_threads;
00875
        p->status = 0;
        p->list = NULL;
00876
00877
        p->cb.on_connected = NULL;
00878
        p->cb.on_connected_arg = NULL;
        p->cb.on_readed = NULL;
00879
08800
        p->cb.on_readed_arg = NULL;
00881
        p->cb.on_wrote = NULL;
        p->cb.on_wrote_arg = NULL;
00882
00883
        p->cb.on_closed = NULL;
        p->cb.on_closed_arg = NULL;
p->cb.on_error = NULL;
00884
00885
00886
        p->cb.on_error_arg = NULL;
        p->next = 0;
p->efds = (int *)malloc (sizeof (int) * p->n_threads);
00887
00888
00889
        if (!p->efds)
00890
          {
00891
            free (p->m_rings);
            free (p);
LOG_END (" ");
00892
00893
00894
            return NULL;
00895
        for (uint32_t i = 0; i < p->n_threads; ++i)
00896
00897
00898
            p->efds[i] = eventfd (0, EFD_NONBLOCK);
00899
             if (p->efds[i] == ERROR_CODE)
00900
00901
                 for (uint32_t j = 0; j < i; ++j)
00902
00903
                     close (p->efds[i]);
00904
00905
                 free (p->efds);
00906
                 free (p->m_rings);
```

```
free (p);
LOG_END (" ");
00908
00909
                return NULL;
00910
00911
00912
        p->rings
            = (struct io_uring *)malloc (sizeof (struct io_uring) * p->n_threads);
00913
00914
        if (!p->rings)
00915
            for (uint32_t j = 0; j < p->n_threads; ++j)
00916
00917
              {
00918
                close (p->efds[j]);
00919
00920
            free (p->efds);
00921
            free (p->m_rings);
            free (p);
LOG_END (" ");
00922
00923
            return NULL;
00924
00925
00926
        for (uint32_t i = 0; i < p->n_threads; ++i)
00927
00928
            memset (&p->rings[i], 0, sizeof (struct io_uring));
            ret = io_uring_queue_init (SAURION_RING_SIZE, &p->rings[i], 0);
00929
00930
            if (ret)
00931
              {
00932
                for (uint32_t j = 0; j < p->n_threads; ++j)
00933
00934
                    close (p->efds[j]);
00935
00936
                free (p->efds);
00937
                free (p->rings);
00938
                free (p->m_rings);
                free (p);
LOG_END (" ");
00939
00940
00941
                return NULL;
00942
00943
00944
       p->pool = threadpool_create (p->n_threads);
00945
        LOG_END (" ");
00946
       return p;
00947 }
```

5.1.3.6 saurion_destroy()

Destroys the saurion structure and frees all associated resources.

This function waits for the event processing to stop, frees the memory used by the saurion structure, and closes any open file descriptors. It ensures that no resources are leaked when the structure is no longer needed.

Parameters

s Pointer to the saurion structure.

Definition at line 1189 of file low saurion.c.

```
01190 {
        pthread mutex lock (&s->status m);
01191
01192
        while (s->status > 0)
01193
01194
             pthread_cond_wait (&s->status_c, &s->status_m);
01195
01196
        pthread_mutex_unlock (&s->status_m);
        threadpool_destroy (s->pool);
for (uint32_t i = 0; i < s->n_threads; ++i)
01197
01198
01199
01200
             io_uring_queue_exit (&s->rings[i]);
01201
             pthread_mutex_destroy (&s->m_rings[i]);
01202
01203
        free (s->m_rings);
```

```
list_free (&s->list);
01205
        for (uint32_t i = 0; i < s->n_threads; ++i)
01206
01207
             close (s->efds[i]);
01208
01209
        free (s->efds);
01210
        if (!s->ss)
        {
01211
01212
            close (s->ss);
01213
01214
       free (s->rings);
       pthread_mutex_destroy (&s->status_m);
pthread_cond_destroy (&s->status_c);
01215
01216
01217
        free (s);
01218 }
```

5.1.3.7 saurion_send()

```
void saurion_send (  struct \ saurion * s, \\ const \ int \ fd, \\ const \ char *const \ msg \ )
```

Sends a message through a socket using io_uring.

This function prepares and sends a message through the specified socket using the io_uring event queue. The message is split into iovec structures for efficient transmission and sent asynchronously.

Parameters

s	Pointer to the saurion structure.
fd	File descriptor of the socket to which the message will be sent.
msg	Pointer to the character string (message) to be sent.

Definition at line 1222 of file low_saurion.c.

```
01223 {
01224 add_write (s, fd, msg, next (s));
01225 }
```

5.1.3.8 saurion_set_socket()

```
\label{eq:control_set_socket} \mbox{ int saurion\_set\_socket (} \\ \mbox{ int } p \mbox{ )}
```

Creates a socket.

Creates and sets a socket, ready for saurion configuration.

Parameters



Returns

result of socket creation.

Definition at line 797 of file low saurion.c.

```
00798 {
00799
        int sock = 0;
00800
        struct sockaddr_in srv_addr;
00801
00802
        sock = socket (PF_INET, SOCK_STREAM, 0);
00803
        if (sock < 1)
00804
00805
             return ERROR CODE;
00806
00807
80800
        int enable = 1;
        if (setsockopt (sock, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof (int)) < 0)</pre>
00809
         {
00810
00811
             return ERROR CODE;
00812
          }
00813
00814
        memset (&srv_addr, 0, sizeof (srv_addr));
        srv_addr.sin_family = AF_INET;
srv_addr.sin_port = htons (p);
00815
00816
        srv_addr.sin_addr.s_addr = htonl (INADDR_ANY);
00817
00818
00819
        if (bind (sock, (const struct sockaddr *)&srv_addr, sizeof (srv_addr)) < 0)</pre>
00820
00821
             return ERROR_CODE;
          }
00822
00823
00824
        if (listen (sock, ACCEPT_QUEUE) < 0)</pre>
00826
             return ERROR_CODE;
00827
00828
00829
        return sock;
00830 }
```

5.1.3.9 saurion start()

Starts event processing in the saurion structure.

This function begins accepting socket connections and handling io_uring events in a loop. It will run continuously until a stop signal is received, allowing the application to manage multiple socket events asynchronously.

Parameters

```
s Pointer to the saurion structure.
```

Returns

int Returns 0 on success, or 1 if an error occurs.

Definition at line 1147 of file low saurion.c.

```
01148 {
01149    threadpool_init (s->pool);
01150    threadpool_add (s->pool, saurion_worker_master, s);
01151    struct saurion_wrapper *ss = NULL;
01152    for (uint32_t i = 1; i < s->n_threads; ++i)
01153    {
01154         ss = (struct saurion_wrapper *)malloc (sizeof (struct saurion_wrapper));
01155         if (!ss)
```

```
{
01157
               return ERROR_CODE;
01158
             }
            ss->s = s;
01159
01160
            ss->sel = i;
            threadpool_add (s->pool, saurion_worker_slave, ss);
01161
01162
01163
        pthread_mutex_lock (&s->status_m);
01164
        while (s->status < (int)s->n_threads)
01165
            pthread_cond_wait (&s->status_c, &s->status_m);
01166
01167
01168
        pthread_mutex_unlock (&s->status_m);
01169
        return SUCCESS_CODE;
01170 }
```

5.1.3.10 saurion_stop()

```
void saurion_stop ( {\tt const\ struct\ saurion*\ s\ )}
```

Stops event processing in the saurion structure.

This function sends a signal to the eventfd, indicating that the event loop should stop. It gracefully shuts down the processing of any remaining events before exiting.

Parameters

```
s Pointer to the saurion structure.
```

Definition at line 1174 of file low_saurion.c.

5.1.3.11 set_request()

Sets up a request and allocates iovec structures for data handling in liburing.

This function configures a request structure that will be used to send or receive data through liburing's submission queues. It allocates the necessary iovec structures to split the data into manageable chunks, and optionally adds a header if specified. The request is inserted into a list tracking active requests for proper memory management and deallocation upon completion.

Parameters

r	Pointer to a pointer to the request structure. If NULL, a new request is created.		
1	Pointer to the list of active requests (Node list) where the request will be inserted		
s	Size of the data to be handled. Adjusted if the header flag (h) is true.		
m	Pointer to the memory block containing the data to be processed.		
h	Header flag. If true, a header (sizeof(uint64_t) + 1) is added to the iovec data.		

Returns

int Returns SUCCESS_CODE on success, or ERROR_CODE on failure (memory allocation issues or insertion failure).

Return values

SUCCESS_CODE	The request was successfully set up and inserted into the list.
ERROR_CODE	Memory allocation failed, or there was an error inserting the request into the list.

Note

The function handles memory allocation for the request and iovec structures, and ensures that the memory is freed properly if an error occurs. Pointers to the iovec blocks (children_ptr) are managed and used for proper memory deallocation.

Definition at line 188 of file low_saurion.c.

```
00190 {
        uint64_t full_size = s;
00192
        if (h)
00193
00194
             full_size += (sizeof (uint64_t) + sizeof (uint8_t));
00195
00196
        size_t amount = full_size / CHUNK_SZ;
        amount = amount + (full_size % CHUNK_SZ == 0 ? 0 : 1);
00197
        struct request *temp = (struct request *)malloc (
    sizeof (struct request) + sizeof (struct iovec) * amount);
00198
00199
00200
        if (!temp)
00201
         {
             return ERROR_CODE;
00202
00203
           }
        if (!*r)
00204
00205
         {
00206
             *r = temp;
             (*r)->prev = NULL;
00207
00208
             (*r) \rightarrow prev\_size = 0;
             (*r)->prev_remain = 0;
00209
00210
             (*r) ->next_iov = 0;
00211
             (*r)->next_offset = 0;
00212
00213
        else
00214
         {
00215
             temp->client_socket = (*r)->client_socket;
             temp->event_type = (*r)->event_type;
temp->prev = (*r)->prev;
00216
00217
             temp->prev_size = (*r)->prev_size;
temp->prev_remain = (*r)->prev_remain;
00218
00219
00220
             temp->next_iov = (*r)->next_iov;
             temp->next_offset = (*r)->next_offset;
00221
00222
             *r = temp;
00223
00224
        struct request *req = *r;
        req->iovec_count = (int) amount;
00225
00226
        void **children_ptr = (void **)malloc (amount * sizeof (void *));
00227
        if (!children_ptr)
00228
00229
             free_request (req, children_ptr, 0);
00230
             return ERROR_CODE;
00231
```

5.2 ThreadPool 23

```
for (size_t i = 0; i < amount; ++i)</pre>
00233
00234
            if (!allocate_iovec (&req->iov[i], amount, i, full_size, children_ptr))
00235
00236
               free_request (req, children_ptr, amount);
                return ERROR_CODE;
00237
00239
            if (!initialize_iovec (&req->iov[i], amount, i, m, s, h))
00240
                free_request (req, children_ptr, amount);
00241
00242
                return ERROR_CODE;
00243
00244
00245
       if (list_insert (l, req, amount, children_ptr))
00246
00247
            free_request (req, children_ptr, amount);
00248
            return ERROR_CODE;
00249
       free (children_ptr);
00250
00251
       return SUCCESS_CODE;
00252 }
```

5.2 ThreadPool

Functions

- struct threadpool * threadpool_create (size_t num_threads)
- struct threadpool * threadpool_create_default (void)
- void threadpool_init (struct threadpool *pool)
- void threadpool_add (struct threadpool *pool, void(*function)(void *), void *argument)
- void threadpool_stop (struct threadpool *pool)
- int threadpool empty (struct threadpool *pool)
- void threadpool_wait_empty (struct threadpool *pool)
- void threadpool_destroy (struct threadpool *pool)

5.2.1 Detailed Description

5.2.2 Function Documentation

5.2.2.1 threadpool add()

Definition at line 175 of file threadpool.c.

```
00178
        LOG_INIT (" ");
        if (pool == NULL || function == NULL)
00179
00180
         {
           LOG_END (" ");
00181
00182
            return;
00183
00184
00185
       struct task *new_task = malloc (sizeof (struct task));
00186
        if (new_task == NULL)
00187
00188
           perror ("Failed to allocate task");
00189
            LOG_END (" ");
```

```
00190
            return;
00191
00192
        new_task->function = function;
new_task->argument = argument;
00193
00194
        new_task->next = NULL;
00195
00196
00197
        pthread_mutex_lock (&pool->queue_lock);
00198
00199
        if (pool->task_queue_head == NULL)
00200
            pool->task_queue_head = new_task;
00201
00202
            pool->task_queue_tail = new_task;
00203
00204
        else
00205
            pool->task_queue_tail->next = new_task;
00206
00207
            pool->task_queue_tail = new_task;
00208
00209
        pthread_cond_signal (&pool->queue_cond);
00210
00211
        pthread_mutex_unlock (&pool->queue_lock);
        LOG_END (" ");
00212
00213 }
```

5.2.2.2 threadpool_create()

Definition at line 32 of file threadpool.c.

```
00033 4
00034
        LOG_INIT (" ");
00035
        struct threadpool *pool = malloc (sizeof (struct threadpool));
00036
        if (pool == NULL)
00037
            perror ("Failed to allocate threadpool"); LOG_END (" ");
00038
00039
00040
            return NULL;
00041
00042
        if (num_threads < 3)</pre>
00043
00044
            num_threads = 3;
00045
00046
        if (num threads > NUM CORES)
00047
00048
            num_threads = NUM_CORES;
00049
00050
00051
        pool->num_threads = num_threads;
00052
        pool->threads = malloc (sizeof (pthread_t) * num_threads);
00053
        if (pool->threads == NULL)
00054
00055
            perror ("Failed to allocate threads array");
            free (pool);
LOG_END (" ");
00056
00057
00058
            return NULL;
00059
00060
00061
        pool->task_queue_head = NULL;
00062
        pool->task_queue_tail = NULL;
00063
        pool->stop = FALSE;
00064
        pool->started = FALSE;
00065
        if (pthread_mutex_init (&pool->queue_lock, NULL) != 0)
00066
00067
00068
            perror ("Failed to initialize mutex");
00069
            free (pool->threads);
00070
            free (pool);
LOG_END (" ");
00071
00072
            return NULL;
00073
00074
00075
        if (pthread_cond_init (&pool->queue_cond, NULL) != 0)
00076
00077
            perror ("Failed to initialize condition variable");
00078
            pthread_mutex_destroy (&pool->queue_lock);
            free (pool->threads);
```

5.2 ThreadPool 25

```
08000
              free (pool);
LOG_END (" ");
00081
00082
               return NULL;
00083
00084
          if (pthread_cond_init (&pool->empty_cond, NULL) != 0)
00085
00087
              perror ("Failed to initialize empty condition variable");
              pthread_mutex_destroy (&pool->queue_lock);
pthread_cond_destroy (&pool->queue_cond);
free (pool->threads);
00088
00089
00090
00091
              free (pool);
LOG_END (" ");
00092
00093
               return NULL;
00094
00095
         LOG_END (" ");
00096
00097
         return pool;
00098 }
```

5.2.2.3 threadpool_create_default()

Definition at line 101 of file threadpool.c.

```
00102 {
00103   return threadpool_create (NUM_CORES);
00104 }
```

5.2.2.4 threadpool_destroy()

Definition at line 274 of file threadpool.c.

```
00275 {
         LOG_INIT (" ");
00276
         if (pool == NULL)
00277
00278
          {
00279
             LOG_END (" ");
00280
             return;
00281
00282
        threadpool_stop (pool);
00283
00284
         pthread_mutex_lock (&pool->queue_lock);
00285
         struct task *task = pool->task_queue_head;
00286
         while (task != NULL)
00287
          {
00288
             struct task *tmp = task;
task = task->next;
00289
00290
             free (tmp);
00291
00292
         pthread_mutex_unlock (&pool->queue_lock);
00293
00294
         pthread_mutex_destroy (&pool->queue_lock);
         pthread_cond_destroy (&pool->queue_cond);
pthread_cond_destroy (&pool->empty_cond);
00295
00296
00297
00298
         free (pool->threads);
        free (pool);
LOG_END (" ");
00299
00300
00301 }
```

5.2.2.5 threadpool_empty()

pthread_mutex_lock (&pool->queue_lock);
int empty = (pool->task_queue_head == NULL);

pthread_mutex_unlock (&pool->queue_lock);

00251 LOG_END (" "); 00252 return empty; 00253 }

5.2.2.6 threadpool_init()

00247

00248

Definition at line 151 of file threadpool.c.

```
00152 {
00153
        LOG_INIT (" ");
00154
        if (pool == NULL || pool->started)
00155
            LOG_END (" ");
00156
00157
            return;
00158
        for (size_t i = 0; i < pool->num_threads; i++)
00159
00160
00161
            if (pthread_create (&pool->threads[i], NULL, threadpool_worker,
00162
                                 (void *)pool)
00163
                != 0)
             {
00164
              perror ("Failed to create thread");
00165
00166
                pool->stop = TRUE;
00167
00168
00169
       pool->started = TRUE;
LOG_END (" ");
00170
00171
00172 }
```

5.2.2.7 threadpool_stop()

```
void threadpool_stop ( {\tt struct\ threadpool\ *\ pool\ )}
```

Definition at line 216 of file threadpool.c.

```
00217 {
       LOG_INIT (" ");
00218
       if (pool == NULL || !pool->started)
00219
00220
           LOG_END (" ");
00222
           return;
00223
00224
       threadpool_wait_empty (pool);
00225
       pthread_mutex_lock (&pool->queue_lock);
00226
00227
       pool->stop = TRUE;
       pthread_cond_broadcast (&pool->queue_cond);
```

5.2 ThreadPool 27

5.2.2.8 threadpool_wait_empty()

```
void threadpool_wait_empty ( {\tt struct\ threadpool\ *\ pool\ )}
```

Definition at line 256 of file threadpool.c.

```
00257 {
        LOG_INIT (" ");
00258
00259
        if (pool == NULL)
00260
           LOG_END (" ");
00261
00262
           return;
00263
00264
        pthread_mutex_lock (&pool->queue_lock);
00265
        while (pool->task_queue_head != NULL)
00266
00267
00268
           pthread_cond_wait (&pool->empty_cond, &pool->queue_lock);
00269
        pthread_mutex_unlock (&pool->queue_lock);
00270
        LOG_END (" ");
00271 }
```

28 Module Documentation

Chapter 6

Class Documentation

6.1 chunk_params Struct Reference

Collaboration diagram for chunk_params:

Public Attributes

- void ** dest
- void * dest_ptr
- size_t dest_off
- struct request * req
- size_t cont_sz
- size_t cont_rem
- size_t max_iov_cont
- size_t curr_iov
- size_t curr_iov_off
- size_t * len

6.1.1 Detailed Description

Definition at line 452 of file low_saurion.c.

6.1.2 Member Data Documentation

6.1.2.1 cont_rem

```
size_t chunk_params::cont_rem
```

Definition at line 459 of file low_saurion.c.

6.1.2.2 cont_sz

size_t chunk_params::cont_sz

Definition at line 458 of file low_saurion.c.

6.1.2.3 curr_iov

size_t chunk_params::curr_iov

Definition at line 461 of file low_saurion.c.

6.1.2.4 curr_iov_off

size_t chunk_params::curr_iov_off

Definition at line 462 of file low_saurion.c.

6.1.2.5 dest

void** chunk_params::dest

Definition at line 454 of file low_saurion.c.

6.1.2.6 dest off

size_t chunk_params::dest_off

Definition at line 456 of file low_saurion.c.

6.1.2.7 dest_ptr

void* chunk_params::dest_ptr

Definition at line 455 of file low_saurion.c.

6.2 Node Struct Reference 31

6.1.2.8 len

```
size_t* chunk_params::len
```

Definition at line 463 of file low_saurion.c.

6.1.2.9 max_iov_cont

```
size_t chunk_params::max_iov_cont
```

Definition at line 460 of file low_saurion.c.

6.1.2.10 req

```
struct request* chunk_params::req
```

Definition at line 457 of file low_saurion.c.

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

• /__w/saurion/saurion/src/low_saurion.c

6.2 Node Struct Reference

Collaboration diagram for Node:

Public Attributes

- void * ptr
- size_t size
- struct Node ** children
- struct Node * next

6.2.1 Detailed Description

Definition at line 6 of file linked_list.c.

6.2.2 Member Data Documentation

6.2.2.1 children

```
struct Node** Node::children
```

Definition at line 10 of file linked_list.c.

6.2.2.2 next

```
struct Node* Node::next
```

Definition at line 11 of file linked_list.c.

6.2.2.3 ptr

```
void* Node::ptr
```

Definition at line 8 of file linked_list.c.

6.2.2.4 size

```
size_t Node::size
```

Definition at line 9 of file linked_list.c.

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

• /__w/saurion/saurion/src/linked_list.c

6.3 request Struct Reference

Public Attributes

- void * prev
- size_t prev_size
- size_t prev_remain
- size_t next_iov
- size_t next_offset
- int event_type
- size_t iovec_count
- int client_socket
- struct iovec iov []

6.3.1 Detailed Description

Definition at line 32 of file low_saurion.c.

6.3.2 Member Data Documentation

6.3.2.1 client_socket

```
int request::client_socket
```

Definition at line 41 of file low_saurion.c.

6.3.2.2 event_type

```
int request::event_type
```

Definition at line 39 of file low_saurion.c.

6.3.2.3 iov

```
struct iovec request::iov[]
```

Definition at line 42 of file low_saurion.c.

6.3.2.4 iovec_count

```
size_t request::iovec_count
```

Definition at line 40 of file low_saurion.c.

6.3.2.5 next_iov

```
size_t request::next_iov
```

Definition at line 37 of file low_saurion.c.

6.3.2.6 next_offset

```
size_t request::next_offset
```

Definition at line 38 of file low_saurion.c.

6.3.2.7 prev

```
void* request::prev
```

Definition at line 34 of file low_saurion.c.

6.3.2.8 prev_remain

```
size_t request::prev_remain
```

Definition at line 36 of file low_saurion.c.

6.3.2.9 prev_size

```
size_t request::prev_size
```

Definition at line 35 of file low_saurion.c.

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

• /__w/saurion/saurion/src/low_saurion.c

6.4 saurion Struct Reference

Main structure for managing io_uring and socket events.

```
#include <low_saurion.h>
```

Collaboration diagram for saurion:

Classes

• struct saurion_callbacks

Structure containing callback functions to handle socket events.

Public Attributes

```
• struct io_uring * rings
```

- pthread_mutex_t * m_rings
- int ss
- int * efds
- struct Node * list
- pthread_mutex_t status_m
- pthread_cond_t status_c
- int status
- struct threadpool * pool
- uint32_t n_threads
- uint32_t next

6.4.1 Detailed Description

Main structure for managing io_uring and socket events.

This structure contains all the necessary data to handle the io_uring event queue and the callbacks for socket events, enabling efficient asynchronous I/O operations.

Definition at line 148 of file low_saurion.h.

6.4.2 Member Data Documentation

6.4.2.1 efds

```
int* saurion::efds
```

Eventfd descriptors used for internal signaling between threads.

Definition at line 157 of file low_saurion.h.

6.4.2.2 list

```
struct Node* saurion::list
```

Linked list for storing active requests.

Definition at line 159 of file low_saurion.h.

6.4.2.3 m_rings

```
pthread_mutex_t* saurion::m_rings
```

Array of mutexes to protect the io_uring rings.

Definition at line 153 of file low saurion.h.

6.4.2.4 n_threads

```
uint32_t saurion::n_threads
```

Number of threads in the thread pool.

Definition at line 169 of file low_saurion.h.

6.4.2.5 next

```
uint32_t saurion::next
```

Index of the next io_uring ring to which an event will be added.

Definition at line 171 of file low_saurion.h.

6.4.2.6 pool

```
struct threadpool* saurion::pool
```

Thread pool for executing tasks in parallel.

Definition at line 167 of file low saurion.h.

6.4.2.7 rings

```
struct io_uring* saurion::rings
```

Array of io_uring structures for managing the event queue.

Definition at line 151 of file low_saurion.h.

6.4.2.8 ss

int saurion::ss

Server socket descriptor for accepting connections.

Definition at line 155 of file low saurion.h.

6.4.2.9 status

int saurion::status

Current status of the structure (e.g., running, stopped).

Definition at line 165 of file low_saurion.h.

6.4.2.10 status_c

```
pthread_cond_t saurion::status_c
```

Condition variable to signal changes in the structure's state.

Definition at line 163 of file low_saurion.h.

6.4.2.11 status_m

```
pthread_mutex_t saurion::status_m
```

Mutex to protect the state of the structure.

Definition at line 161 of file low_saurion.h.

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

• /__w/saurion/saurion/include/low_saurion.h

6.5 Saurion Class Reference

#include <saurion.hpp>

Collaboration diagram for Saurion:

Public Types

- using ConnectedCb = void(*)(const int, void *)
- using ReadedCb = void(*)(const int, const void *const, const ssize_t, void *)
- using WroteCb = void(*)(const int, void *)
- using ClosedCb = void(*)(const int, void *)
- using ErrorCb = void(*)(const int, const char *const, const ssize_t, void *)

Public Member Functions

- · Saurion (const uint32_t thds, const int sck) noexcept
- ∼Saurion ()
- Saurion (const Saurion &)=delete
- Saurion (Saurion &&)=delete
- Saurion & operator= (const Saurion &)=delete
- Saurion & operator= (Saurion &&)=delete
- void init () noexcept
- void stop () const noexcept
- Saurion * on_connected (ConnectedCb ncb, void *arg) noexcept
- Saurion * on_readed (ReadedCb ncb, void *arg) noexcept
- Saurion * on_wrote (WroteCb ncb, void *arg) noexcept
- Saurion * on_closed (ClosedCb ncb, void *arg) noexcept
- Saurion * on_error (ErrorCb ncb, void *arg) noexcept
- · void send (const int fd, const char *const msg) noexcept

Private Attributes

struct saurion * s

6.5.1 Detailed Description

Definition at line 7 of file saurion.hpp.

6.5.2 Member Typedef Documentation

6.5.2.1 ClosedCb

```
using Saurion::ClosedCb = void (*) (const int, void *)
```

Definition at line 14 of file saurion.hpp.

6.5.2.2 ConnectedCb

```
using Saurion::ConnectedCb = void (*) (const int, void *)
```

Definition at line 10 of file saurion.hpp.

6.5.2.3 ErrorCb

```
using Saurion::ErrorCb = void (*) (const int, const char *const, const ssize_t, void *)
```

Definition at line 15 of file saurion.hpp.

6.5.2.4 ReadedCb

```
using Saurion::ReadedCb = void (*) (const int, const void *const, const ssize_t, void *)
```

Definition at line 11 of file saurion.hpp.

6.5.2.5 WroteCb

```
using Saurion::WroteCb = void (*) (const int, void *)
```

Definition at line 13 of file saurion.hpp.

6.5.3 Constructor & Destructor Documentation

6.5.3.1 Saurion() [1/3]

Definition at line 5 of file saurion.cpp.

```
00006 {
00007    this->s = saurion_create (thds);
00008    if (!this->s)
00009    {
00010         return;
00011    }
00012    this->s->ss = sck;
00013 }
```

6.5.3.2 ~Saurion()

```
Saurion::~Saurion ()
```

Definition at line 15 of file saurion.cpp. 00015 { saurion_destroy (this->s); }

6.5.3.3 Saurion() [2/3]

6.5.3.4 Saurion() [3/3]

6.5.4 Member Function Documentation

6.5.4.1 init()

```
void Saurion::init ( ) [noexcept]
```

Definition at line 18 of file saurion.cpp.

6.5.4.2 on_closed()

Definition at line 57 of file saurion.cpp.

6.5.4.3 on_connected()

```
Saurion * Saurion::on_connected (
             Saurion::ConnectedCb ncb,
              void * arg ) [noexcept]
Definition at line 33 of file saurion.cpp.
```

```
00035
        s->cb.on_connected = ncb;
00036
       s->cb.on_connected_arg = arg;
00037
       return this;
00038 }
```

6.5.4.4 on_error()

```
Saurion * Saurion::on_error (
            Saurion::ErrorCb ncb,
            void * arg ) [noexcept]
```

Definition at line 65 of file saurion.cpp.

```
00066 {
00067 s->cb.on_error = ncb;
00068 s->cb.on_error_arg = arg;
00069 return this;
00070 }
```

6.5.4.5 on_readed()

```
Saurion * Saurion::on_readed (
            Saurion::ReadedCb ncb,
            void * arg ) [noexcept]
```

Definition at line 41 of file saurion.cpp.

```
00042 {
00043
       s->cb.on_readed = ncb;
00044
       s->cb.on_readed_arg = arg;
00045
       return this;
00046 }
```

6.5.4.6 on_wrote()

```
Saurion * Saurion::on_wrote (
            Saurion::WroteCb ncb,
            void * arg ) [noexcept]
```

Definition at line 49 of file saurion.cpp.

```
00050 {
00051
       s->cb.on_wrote = ncb;
00052
       s->cb.on_wrote_arg = arg;
00053 return this;
00054 }
```

6.5.4.7 operator=() [1/2]

6.5.4.8 operator=() [2/2]

6.5.4.9 send()

Definition at line 73 of file saurion.cpp.

6.5.4.10 stop()

```
void Saurion::stop ( ) const [noexcept]
```

Definition at line 27 of file saurion.cpp.

6.5.5 Member Data Documentation

6.5.5.1 s

```
struct saurion* Saurion::s [private]
```

Definition at line 38 of file saurion.hpp.

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

- /__w/saurion/saurion/include/saurion.hpp
- /__w/saurion/saurion/src/saurion.cpp

6.6 saurion::saurion callbacks Struct Reference

Structure containing callback functions to handle socket events.

```
#include <low_saurion.h>
```

Public Attributes

void(* on connected)(const int fd, void *arg)

Callback for handling new connections.

- void * on_connected_arg
- void(* on_readed)(const int fd, const void *const content, const ssize_t len, void *arg)

Callback for handling read events.

- void * on_readed_arg
- void(* on_wrote)(const int fd, void *arg)

Callback for handling write events.

- void * on_wrote_arg
- void(* on_closed)(const int fd, void *arg)

Callback for handling socket closures.

- void * on_closed_arg
- void(* on_error)(const int fd, const char *const content, const ssize_t len, void *arg)

Callback for handling error events.

void * on_error_arg

6.6.1 Detailed Description

Structure containing callback functions to handle socket events.

This structure holds pointers to callback functions for handling events such as connection establishment, reading, writing, closing, and errors on sockets. Each callback has an associated argument pointer that can be passed along when the callback is invoked.

Definition at line 181 of file low saurion.h.

6.6.2 Member Data Documentation

6.6.2.1 on_closed

```
void(* saurion::saurion_callbacks::on_closed) (const int fd, void *arg)
```

Callback for handling socket closures.

Parameters

fd	File descriptor of the closed socket.
arg	Additional user-provided argument.

Definition at line 221 of file low_saurion.h.

6.6.2.2 on_closed_arg

void* saurion::saurion_callbacks::on_closed_arg

Additional argument for the close callback.

Definition at line 223 of file low_saurion.h.

6.6.2.3 on_connected

 $\verb|void(* saurion::saurion_callbacks::on_connected)| (const int fd, \verb|void *| arg)| \\$

Callback for handling new connections.

Parameters

fd	File descriptor of the connected socket.
arg	Additional user-provided argument.

Definition at line 189 of file low_saurion.h.

6.6.2.4 on_connected_arg

 $\verb"void* saurion::saurion_callbacks::on_connected_arg"$

Additional argument for the connection callback.

Definition at line 191 of file low_saurion.h.

6.6.2.5 on_error

void(* saurion::saurion_callbacks::on_error) (const int fd, const char *const content, const
ssize_t len, void *arg)

Callback for handling error events.

Parameters

fd	File descriptor of the socket where the error occurred.
content	Pointer to the error message.
len	Length of the error message.
arg	Additional user-provided argument.

Generated by Doxygen

Definition at line 233 of file low_saurion.h.

6.6.2.6 on_error_arg

void* saurion::saurion_callbacks::on_error_arg

Additional argument for the error callback.

Definition at line 236 of file low_saurion.h.

6.6.2.7 on_readed

void(* saurion::saurion_callbacks::on_readed) (const int fd, const void *const content, const
ssize_t len, void *arg)

Callback for handling read events.

Parameters

fd	File descriptor of the socket.
content	Pointer to the data that was read.
len	Length of the data that was read.
arg	Additional user-provided argument.

Definition at line 201 of file low_saurion.h.

6.6.2.8 on readed arg

void* saurion::saurion_callbacks::on_readed_arg

Additional argument for the read callback.

Definition at line 204 of file low_saurion.h.

6.6.2.9 on_wrote

void(* saurion::saurion_callbacks::on_wrote) (const int fd, void *arg)

Callback for handling write events.

Parameters

fd	File descriptor of the socket.
arg	Additional user-provided argument.

Definition at line 212 of file low_saurion.h.

6.6.2.10 on_wrote_arg

```
void* saurion::saurion_callbacks::on_wrote_arg
```

Additional argument for the write callback.

Definition at line 213 of file low_saurion.h.

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

• /__w/saurion/saurion/include/low_saurion.h

6.7 saurion_callbacks Struct Reference

Structure containing callback functions to handle socket events.

```
#include <low_saurion.h>
```

Public Attributes

- void(* on_connected)(const int fd, void *arg)
 - Callback for handling new connections.
- void * on_connected_arg
- void(* on_readed)(const int fd, const void *const content, const ssize_t len, void *arg)

Callback for handling read events.

- void * on_readed_arg
- void(* on_wrote)(const int fd, void *arg)

Callback for handling write events.

- void * on_wrote_arg
- void(* on_closed)(const int fd, void *arg)

Callback for handling socket closures.

- void * on_closed_arg
- void(* on_error)(const int fd, const char *const content, const ssize_t len, void *arg)

Callback for handling error events.

void * on_error_arg

6.7.1 Detailed Description

Structure containing callback functions to handle socket events.

This structure holds pointers to callback functions for handling events such as connection establishment, reading, writing, closing, and errors on sockets. Each callback has an associated argument pointer that can be passed along when the callback is invoked.

Definition at line 31 of file low_saurion.h.

6.7.2 Member Data Documentation

6.7.2.1 on_closed

```
void(* saurion_callbacks::on_closed) (const int fd, void *arg)
```

Callback for handling socket closures.

Parameters

fd	File descriptor of the closed socket.
arg	Additional user-provided argument.

Definition at line 71 of file low_saurion.h.

6.7.2.2 on_closed_arg

```
void* saurion_callbacks::on_closed_arg
```

Additional argument for the close callback.

Definition at line 73 of file low_saurion.h.

6.7.2.3 on_connected

```
void(* saurion_callbacks::on_connected) (const int fd, void *arg)
```

Callback for handling new connections.

Parameters

fd	File descriptor of the connected socket.
arg	Additional user-provided argument.

Generated by Doxygen

Definition at line 39 of file low_saurion.h.

6.7.2.4 on_connected_arg

```
void* saurion_callbacks::on_connected_arg
```

Additional argument for the connection callback.

Definition at line 41 of file low_saurion.h.

6.7.2.5 on_error

 $\label{local_const_con$

Callback for handling error events.

Parameters

fd	File descriptor of the socket where the error occurred.
content	Pointer to the error message.
len	Length of the error message.
arg	Additional user-provided argument.

Definition at line 83 of file low saurion.h.

6.7.2.6 on_error_arg

```
void* saurion_callbacks::on_error_arg
```

Additional argument for the error callback.

Definition at line 86 of file low_saurion.h.

6.7.2.7 on_readed

 $\label{local_const_con$

Callback for handling read events.

Parameters

fd	File descriptor of the socket.
content	Pointer to the data that was read.
len	Length of the data that was read.
arg	Additional user-provided argument.

Definition at line 51 of file low_saurion.h.

6.7.2.8 on_readed_arg

void* saurion_callbacks::on_readed_arg

Additional argument for the read callback.

Definition at line 54 of file low_saurion.h.

6.7.2.9 on_wrote

void(* saurion_callbacks::on_wrote) (const int fd, void *arg)

Callback for handling write events.

Parameters

fd	File descriptor of the socket.
arg	Additional user-provided argument.

Definition at line 62 of file low_saurion.h.

6.7.2.10 on_wrote_arg

void* saurion_callbacks::on_wrote_arg

Additional argument for the write callback.

Definition at line 63 of file low_saurion.h.

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

• /__w/saurion/saurion/include/low_saurion.h

6.8 saurion_wrapper Struct Reference

Collaboration diagram for saurion_wrapper:

Public Attributes

- struct saurion * s
- uint32_t sel

6.8.1 Detailed Description

Definition at line 50 of file low_saurion.c.

6.8.2 Member Data Documentation

6.8.2.1 s

```
struct saurion* saurion_wrapper::s
```

Definition at line 52 of file low_saurion.c.

6.8.2.2 sel

```
uint32_t saurion_wrapper::sel
```

Definition at line 53 of file low_saurion.c.

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

• /__w/saurion/saurion/src/low_saurion.c

6.9 task Struct Reference

Collaboration diagram for task:

Public Attributes

- void(* function)(void *)
- void * argument
- struct task * next

6.9.1 Detailed Description

Definition at line 11 of file threadpool.c.

6.9.2 Member Data Documentation

6.9.2.1 argument

```
void* task::argument
```

Definition at line 14 of file threadpool.c.

6.9.2.2 function

```
void(* task::function) (void *)
```

Definition at line 13 of file threadpool.c.

6.9.2.3 next

```
struct task* task::next
```

Definition at line 15 of file threadpool.c.

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

• /__w/saurion/saurion/src/threadpool.c

6.10 threadpool Struct Reference

Collaboration diagram for threadpool:

Public Attributes

- pthread_t * threads
- size_t num_threads
- struct task * task_queue_head
- struct task * task_queue_tail
- pthread_mutex_t queue_lock
- pthread_cond_t queue_cond
- pthread_cond_t empty_cond
- int stop
- · int started

6.10.1 Detailed Description

Definition at line 18 of file threadpool.c.

6.10.2 Member Data Documentation

6.10.2.1 empty_cond

pthread_cond_t threadpool::empty_cond

Definition at line 26 of file threadpool.c.

6.10.2.2 num threads

size_t threadpool::num_threads

Definition at line 21 of file threadpool.c.

6.10.2.3 queue_cond

pthread_cond_t threadpool::queue_cond

Definition at line 25 of file threadpool.c.

6.10.2.4 queue_lock

pthread_mutex_t threadpool::queue_lock

Definition at line 24 of file threadpool.c.

6.10.2.5 started

int threadpool::started

Definition at line 28 of file threadpool.c.

6.10.2.6 stop

int threadpool::stop

Definition at line 27 of file threadpool.c.

6.10.2.7 task_queue_head

```
struct task* threadpool::task_queue_head
```

Definition at line 22 of file threadpool.c.

6.10.2.8 task_queue_tail

```
struct task* threadpool::task_queue_tail
```

Definition at line 23 of file threadpool.c.

6.10.2.9 threads

```
pthread_t* threadpool::threads
```

Definition at line 20 of file threadpool.c.

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

• /__w/saurion/saurion/src/threadpool.c

Chapter 7

File Documentation

7.1 /_w/saurion/saurion/include/linked_list.h File Reference

```
#include <stddef.h>
```

Include dependency graph for linked_list.h: This graph shows which files directly or indirectly include this file:

Functions

- int list_insert (struct Node **head, void *ptr, size_t amount, void **children)
- void list_delete_node (struct Node **head, const void *const ptr)
- void list_free (struct Node **head)

7.1.1 Function Documentation

7.1.1.1 list_delete_node()

Definition at line 106 of file linked_list.c.

```
00107 {
00108
       pthread_mutex_lock (&list_mutex);
       struct Node *current = *head;
struct Node *prev = NULL;
00109
00110
00111
00112
        if (current && current->ptr == ptr)
00113
            *head = current->next;
00114
00115
            free_node (current);
            pthread_mutex_unlock (&list_mutex);
00116
00117
            return;
00118
00119
00120
       while (current && current->ptr != ptr)
00121
00122
            prev = current;
00123
            current = current->next;
00124
```

56 File Documentation

```
00125
00126
       if (!current)
00127
           pthread_mutex_unlock (&list_mutex);
00128
00129
            return;
00130
00131
00132
       prev->next = current->next;
00133
       free_node (current);
       pthread_mutex_unlock (&list_mutex);
00134
00135 }
```

7.1.1.2 list_free()

Definition at line 138 of file linked_list.c.

```
00139 {
00140
         pthread_mutex_lock (&list_mutex);
        struct Node *current = *head;
struct Node *next;
00141
00142
00143
        while (current)
00145
         {
00146
           next = current->next;
            free_node (current);
current = next;
00147
00148
00149
00150
00151
        *head = NULL;
00152
        pthread_mutex_unlock (&list_mutex);
00153 }
```

7.1.1.3 list_insert()

Definition at line 65 of file linked_list.c.

```
00066 {
00067
       struct Node *new_node = create_node (ptr, amount, children);
00068
       if (!new_node)
        {
00069
00070
           return 1:
00071
00072
       pthread_mutex_lock (&list_mutex);
00073
        if (!*head)
00074
        {
           *head = new_node;
00075
00076
           pthread_mutex_unlock (&list_mutex);
          return 0;
00077
00078
00079
       struct Node *temp = *head;
08000
       while (temp->next)
00081
00082
           temp = temp->next;
00083
00084
       temp->next = new_node;
00085
       pthread_mutex_unlock (&list_mutex);
00086
       return 0;
00087 }
```

7.2 linked_list.h 57

7.2 linked_list.h

Go to the documentation of this file.

```
00001 #ifndef LINKED_LIST_H 00002 #define LINKED_LIST_H
00003
00004 #ifdef __cplusplus
00005 extern "C"
00006 {
00007 #endif
80000
00009 #include <stddef.h>
00010
00011
       struct Node;
00012
00013 int list_insert (struct Node **head, void *ptr, size_t amount,
00014
                           void **children);
00015
00016 void list_delete_node (struct Node **head, const void *const ptr);
00017
00018
       void list_free (struct Node **head);
00019
00020 #ifdef __cplusplus
00021
00022 #endif
00023
00024 #endif // !LINKED_LIST_H
```

7.3 /_w/saurion/saurion/include/low_saurion.h File Reference

```
#include <pthread.h>
#include <stdint.h>
#include <sys/types.h>
```

Include dependency graph for low_saurion.h: This graph shows which files directly or indirectly include this file:

Classes

· struct saurion

Main structure for managing io_uring and socket events.

• struct saurion::saurion_callbacks

Structure containing callback functions to handle socket events.

· struct saurion callbacks

Structure containing callback functions to handle socket events.

Macros

- #define _POSIX_C_SOURCE 200809L
- #define PACKING_SZ 32

Defines the memory alignment size for structures in the saurion class.

58 File Documentation

Functions

```
int saurion_set_socket (int p)
```

Creates a socket.

struct saurion * saurion_create (uint32_t n_threads)

Creates an instance of the saurion structure.

int saurion_start (struct saurion *s)

Starts event processing in the saurion structure.

void saurion_stop (const struct saurion *s)

Stops event processing in the saurion structure.

void saurion_destroy (struct saurion *s)

Destroys the saurion structure and frees all associated resources.

void saurion_send (struct saurion *s, const int fd, const char *const msg)

Sends a message through a socket using io_uring.

Variables

```
    void(* on_connected )(const int fd, void *arg)
```

Callback for handling new connections.

- void * on_connected_arg
- void(* on_readed)(const int fd, const void *const content, const ssize_t len, void *arg)

Callback for handling read events.

- void * on readed arg
- void(* on_wrote)(const int fd, void *arg)

Callback for handling write events.

- void * on_wrote_arg
- void(* on_closed)(const int fd, void *arg)

Callback for handling socket closures.

- void * on_closed_arg
- void(* on_error)(const int fd, const char *const content, const ssize_t len, void *arg)

Callback for handling error events.

- void * on_error_arg
- struct io_uring * rings
- pthread_mutex_t * m_rings
- int ss
- int * efds
- struct Node * list
- pthread_mutex_t status_m
- pthread_cond_t status_c
- · int status
- struct threadpool * pool
- uint32_t n_threads
- uint32_t next

7.3.1 Variable Documentation

7.3.1.1 efds

```
int* efds
```

Eventfd descriptors used for internal signaling between threads.

Definition at line 7 of file low_saurion.h.

7.3.1.2 list

```
struct Node* list
```

Linked list for storing active requests.

Definition at line 9 of file low_saurion.h.

7.3.1.3 m_rings

```
pthread_mutex_t* m_rings
```

Array of mutexes to protect the io_uring rings.

Definition at line 3 of file low_saurion.h.

7.3.1.4 n_threads

```
uint32_t n_threads
```

Number of threads in the thread pool.

Definition at line 19 of file low_saurion.h.

7.3.1.5 next

```
uint32_t next
```

Index of the next io_uring ring to which an event will be added.

Definition at line 21 of file low_saurion.h.

7.3.1.6 on_closed

Callback for handling socket closures.

60 File Documentation

Parameters

fd	File descriptor of the closed socket.
arg	Additional user-provided argument.

Definition at line 38 of file low_saurion.h.

7.3.1.7 on_closed_arg

```
void * on_closed_arg
```

Additional argument for the close callback.

Definition at line 40 of file low_saurion.h.

7.3.1.8 on_connected

Callback for handling new connections.

Parameters

fd	File descriptor of the connected socket.
arg	Additional user-provided argument.

Definition at line 6 of file low_saurion.h.

7.3.1.9 on_connected_arg

```
void * on_connected_arg
```

Additional argument for the connection callback.

Definition at line 8 of file low_saurion.h.

7.3.1.10 on_error

Callback for handling error events.

Parameters

fd	File descriptor of the socket where the error occurred.
content	Pointer to the error message.
len	Length of the error message.
arg	Additional user-provided argument.

Definition at line 50 of file low_saurion.h.

7.3.1.11 on_error_arg

```
void * on_error_arg
```

Additional argument for the error callback.

Definition at line 53 of file low_saurion.h.

7.3.1.12 on_readed

Callback for handling read events.

Parameters

fd	File descriptor of the socket.
content	Pointer to the data that was read.
len	Length of the data that was read.
arg	Additional user-provided argument.

Definition at line 18 of file low_saurion.h.

File Documentation

7.3.1.13 on_readed_arg

```
void * on_readed_arg
```

Additional argument for the read callback.

Definition at line 21 of file low_saurion.h.

7.3.1.14 on_wrote

Callback for handling write events.

Parameters

fd	File descriptor of the socket.
arg	Additional user-provided argument.

Definition at line 29 of file low_saurion.h.

7.3.1.15 on_wrote_arg

```
void * on_wrote_arg
```

Additional argument for the write callback.

Definition at line 30 of file low_saurion.h.

7.3.1.16 pool

```
struct threadpool* pool
```

Thread pool for executing tasks in parallel.

Definition at line 17 of file low_saurion.h.

7.3.1.17 rings

```
struct io_uring* rings
```

Array of io_uring structures for managing the event queue.

Definition at line 1 of file low_saurion.h.

7.3.1.18 ss

int ss

Server socket descriptor for accepting connections.

Definition at line 5 of file low_saurion.h.

7.3.1.19 status

int status

Current status of the structure (e.g., running, stopped).

Definition at line 15 of file low_saurion.h.

7.3.1.20 status_c

```
pthread_cond_t status_c
```

Condition variable to signal changes in the structure's state.

Definition at line 13 of file low_saurion.h.

7.3.1.21 status_m

```
{\tt pthread\_mutex\_t\ status\_m}
```

Mutex to protect the state of the structure.

Definition at line 11 of file low_saurion.h.

7.4 low saurion.h

```
Go to the documentation of this file.
```

```
00104 #ifndef LOW_SAURION_H
00105 #define LOW_SAURION_H
00106
00107 #define _POSIX_C_SOURCE 200809L
00109 #include <pthread.h>
                             // for pthread_mutex_t, pthread_cond_t
00110 #include <stdint.h> // for uint32_t
00111 #include <sys/types.h> // for ssize_t
00112
00113 #ifdef .
               _cplusplus
00114 extern "C"
00115 {
00116 #endif
00117
00139 #define PACKING_SZ 32
00140
       struct saurion
00149
00151
        struct io_uring *rings;
00153
         pthread_mutex_t *m_rings;
00155
         int ss;
int *efds;
00157
00159
         struct Node *list;
00161
          pthread_mutex_t status_m;
00163
          pthread_cond_t status_c;
00165
          int status;
00167
          struct threadpool *pool;
00169
          uint32_t n_threads;
00171
          uint32_t next;
00172
00181
          struct saurion_callbacks
00182
00189
            void (*on_connected) (const int fd, void *arg);
00191
            void *on_connected_arg;
00192
            void (*on_readed) (const int fd, const void *const content,
00202
                                const ssize_t len, void *arg);
00204
            void *on_readed_arg;
00205
00212
            void (*on_wrote) (const int fd, void *arg);
00213
            void *on_wrote_arg;
00221
            void (*on_closed) (const int fd, void *arg);
00223
            void *on_closed_arg;
00224
00233
            void (*on_error) (const int fd, const char *const content,
00234
                               const ssize_t len, void *arg);
00236
            void *on error arg;
00237
          } __attribute__ ((aligned (PACKING_SZ))) cb;
       } __attribute__ ((aligned (PACKING_SZ)));
00238
00239
00249
       int saurion_set_socket (int p);
00250
00263
       [[nodiscard]]
00264
       struct saurion *saurion_create (uint32_t n_threads);
00265
00278
       int saurion_start (struct saurion *s);
00279
00280
00291
       void saurion_stop (const struct saurion *s);
00292
00305
       void saurion_destroy (struct saurion *s);
00306
00319
       void saurion_send (struct saurion *s, const int fd, const char *const msg);
00320
00321 #ifdef __cplusplus
00322 }
00323 #endif
00324
00325 #endif // !LOW_SAURION_H
```

7.5 /_w/saurion/saurion/include/low_saurion_secret.h File Reference

```
#include <bits/types/struct_iovec.h>
#include <stddef.h>
```

00326

```
#include <stdint.h>
```

Include dependency graph for low_saurion_secret.h:

Functions

- int allocate iovec (struct iovec *iov, size t amount, size t pos, size t size, void **chd ptr)
- int initialize_iovec (struct iovec *iov, size_t amount, size_t pos, const void *msg, size_t size, uint8_t h)

Initializes a specified iovec structure with a message fragment.

• int set_request (struct request **r, struct Node **I, size_t s, const void *m, uint8_t h)

Sets up a request and allocates iovec structures for data handling in liburing.

int read_chunk (void **dest, size_t *len, struct request *const req)

Reads a message chunk from the request's iovec buffers, handling messages that may span multiple iovec entries.

void free_request (struct request *req, void **children_ptr, size_t amount)

7.6 low saurion secret.h

```
Go to the documentation of this file.
```

```
00001 #ifndef LOW_SAURION_SECRET_H
00002 #define LOW_SAURION_SECRET_H
00003
00004 #include <bits/types/struct_iovec.h>
00005 #include <stddef.h>
00006 #include <stdint.h>
00007
00008 #ifdef __cplusplus
00009 extern "C" {
00010 #endif
00015 #pragma GCC diagnostic push
00016 #pragma GCC diagnostic ignored "-Wpedantic"
00017 struct request {
00018 void *prev;
00019 size_t prev_size;
00020 size_t prev_remain;
00021 size_t next_lov,
00022 size_t next_offset;
00023 int event_type;
00024 size_t iovec_count;
00025 int client_socket;
00026 struct iovec iov[];
00028 #pragma GCC diagnostic pop
00062 [[nodiscard]]
00063 int allocate_iovec(struct iovec *iov, size_t amount, size_t pos, size_t size, void **chd_ptr);
00064
00097 [[nodiscard]]
00098 int initialize_iovec(struct iovec *iov, size_t amount, size_t pos, const void *msg, size_t size,
00099
00100
00127 [[nodiscard]]
00128 int set_request(struct request **r, struct Node **1, size_t s, const void *m, uint8_t h);
00129
00166 int read_chunk(void **dest, size_t *len, struct request *const req);
00167
00168 void free_request(struct request *req, void **children_ptr, size_t amount);
00172 #ifdef __cplusplus
00173 }
00174 #endif
00176 #endif // !LOW_SAURION_SECRET_H
```

7.7 /__w/saurion/saurion/include/saurion.hpp File Reference

```
#include <stdint.h>
#include <sys/types.h>
```

Include dependency graph for saurion.hpp: This graph shows which files directly or indirectly include this file:

Classes

class Saurion

7.8 saurion.hpp

Go to the documentation of this file.

```
00001 #ifndef SAURION_HPF
00002 #define SAURION_HPP
00003
00004 #include <stdint.h> // for uint32_t
00005 #include <sys/types.h> // for ssize_t
00006
00007 class Saurion
00008 {
00009 public:
00010 using ConnectedCb = void (*) (const int, void *);
        using ReadedCb
00011
        = void (*) (const int, const void *const, const ssize_t, void *);
using WroteCb = void (*) (const int, void *);
00012
00013
        using ClosedCb = void (*) (const int, void *);
00015
        using ErrorCb
00016
              = void (*) (const int, const char *const, const ssize_t, void *);
00017
00018
        explicit Saurion (const uint32_t thds, const int sck) noexcept;
00019
        ~Saurion ();
00020
00021
        Saurion (const Saurion &) = delete;
00022
        Saurion (Saurion &&) = delete;
00023
        Saurion & operator= (const Saurion &) = delete;
00024
        Saurion & operator= (Saurion & &) = delete;
00025
00026
        void init () noexcept;
00027
        void stop () const noexcept;
00028
00029
        Saurion *on_connected (ConnectedCb ncb, void *arg) noexcept;
00030
        Saurion *on_readed (ReadedCb ncb, void *arg) noexcept;
        Saurion *on_wrote (WroteCb ncb, void *arg) noexcept;
Saurion *on_closed (ClosedCb ncb, void *arg) noexcept;
00031
00032
00033
        Saurion *on_error (ErrorCb ncb, void *arg) noexcept;
00034
00035
        void send (const int fd, const char *const msg) noexcept;
00036
00037 private:
00038
        struct saurion *s;
00039 };
00040
00041 #endif // !SAURION_HPP
```

7.9 /_w/saurion/saurion/include/threadpool.h File Reference

#include <stddef.h>

Include dependency graph for threadpool.h: This graph shows which files directly or indirectly include this file:

Functions

- struct threadpool * threadpool_create (size_t num_threads)
- struct threadpool * threadpool_create_default (void)
- void threadpool_init (struct threadpool *pool)
- void threadpool_add (struct threadpool *pool, void(*function)(void *), void *argument)
- void threadpool_stop (struct threadpool *pool)
- int threadpool_empty (struct threadpool *pool)
- void threadpool wait empty (struct threadpool *pool)
- void threadpool_destroy (struct threadpool *pool)

7.10 threadpool.h

7.10 threadpool.h

Go to the documentation of this file.

```
00001
00006 #ifndef THREADPOOL_H
00007 #define THREADPOOL_H
80000
00009 #include <stddef.h> // for size_t
00010
00011 #ifdef __cplusplus 00012 extern "C"
00013 {
00014 #endif
00015
00016
       struct threadpool;
00017
00018
       struct threadpool *threadpool_create (size_t num_threads);
00019
       struct threadpool *threadpool_create_default (void);
00021
00022
        void threadpool_init (struct threadpool *pool);
00023
00024
        void threadpool_add (struct threadpool *pool, void (*function) (void *),
00025
                              void *argument);
00026
00027
        void threadpool_stop (struct threadpool *pool);
00028
00029
       int threadpool_empty (struct threadpool *pool);
00030
00031
       void threadpool_wait_empty (struct threadpool *pool);
00032
00033
       void threadpool_destroy (struct threadpool *pool);
00034
00035 #ifdef __cplusplus
00036
00037 #endif
00038
00039 #endif // !THREADPOOL_H
00040
```

7.11 /__w/saurion/saurion/src/linked_list.c File Reference

```
#include "linked_list.h"
#include <pthread.h>
#include <stdlib.h>
Include dependency graph for linked_list.c:
```

Classes

struct Node

Functions

- struct Node * create_node (void *ptr, size_t amount, void **children)
- int list_insert (struct Node **head, void *ptr, size_t amount, void **children)
- void free_node (struct Node *current)
- void list_delete_node (struct Node **head, const void *const ptr)
- void list_free (struct Node **head)

Variables

pthread_mutex_t list_mutex = PTHREAD_MUTEX_INITIALIZER

7.11.1 Function Documentation

7.11.1.1 create_node()

```
struct Node * create_node (
               void * ptr,
               size_t amount,
               void ** children )
Definition at line 17 of file linked list.c.
00018 {
00019
        struct Node *new_node = (struct Node *)malloc (sizeof (struct Node));
        if (!new_node)
00021
00022
             return NULL;
00023
        new_node->ptr = ptr;
new_node->size = amount;
00024
00025
00026
        new_node->children = NULL;
00027
        if (amount <= 0)</pre>
00028
00029
            new_node->next = NULL;
00030
            return new_node;
00031
00032
        new_node->children
00033
             = (struct Node **) malloc (sizeof (struct Node *) * amount);
00034
        if (!new_node->children)
00035
            free (new_node);
00036
00037
             return NULL;
00038
00039
        for (size_t i = 0; i < amount; ++i)</pre>
00040
00041
             new_node->children[i] = (struct Node *)malloc (sizeof (struct Node));
00042
             if (!new_node->children[i])
00043
00044
00045
                 for (size_t j = 0; j < i; ++j)
00046
00047
                     free (new_node->children[j]);
00048
00049
                 free (new_node);
00050
                 return NULL:
00051
00052
00053
        for (size_t i = 0; i < amount; ++i)</pre>
00054
00055
            new_node->children[i]->size = 0;
            new_node->children[i]->next = NULL;
new_node->children[i]->ptr = children[i];
00056
00057
00058
            new_node->children[i]->children = NULL;
00059
00060
        new_node->next = NULL;
00061
        return new_node;
00062 }
```

7.11.1.2 free_node()

00095

7.11.1.3 list delete node()

Definition at line 106 of file linked list.c.

```
00107
00108
        pthread_mutex_lock (&list_mutex);
        struct Node *current = *head;
00110
        struct Node *prev = NULL;
00111
00112
       if (current && current->ptr == ptr)
00113
        {
           *head = current->next;
00114
            free_node (current);
00115
00116
           pthread_mutex_unlock (&list_mutex);
00117
            return;
00118
        }
00119
       while (current && current->ptr != ptr)
00120
00121
        {
00122
           prev = current;
00123
            current = current->next;
        }
00124
00125
00126
       if (!current)
00127
        {
         pthread_mutex_unlock (&list_mutex);
00128
00129
            return;
00130
00131
00132 prev->next = current->next;
00133 free_node (current);
00134
       pthread_mutex_unlock (&list_mutex);
00135 }
```

7.11.1.4 list_free()

Definition at line 138 of file linked list.c.

```
00139 {
00140
       pthread_mutex_lock (&list_mutex);
00141
       struct Node *current = *head;
00142
       struct Node *next;
00143
00144
       while (current)
00145
       {
00146
          next = current->next;
00147
           free_node (current);
00148
           current = next;
        }
00149
00150
00151
       *head = NULL;
00152 pthread_mutex_unlock (&list_mutex);
00153 }
```

7.11.1.5 list_insert()

```
int list_insert (
              struct Node ** head,
              void * ptr,
              size_t amount,
              void ** children )
Definition at line 65 of file linked list.c.
00066
00067
       struct Node *new_node = create_node (ptr, amount, children);
00068
       if (!new_node)
00069
00070
           return 1;
00071
00072
       pthread_mutex_lock (&list_mutex);
00073
       if (!*head)
        {
00074
00075
           *head = new_node;
00076
           pthread_mutex_unlock (&list_mutex);
00077
          return 0;
00078
00079
       struct Node *temp = *head;
08000
       while (temp->next)
       {
00081
00082
           temp = temp->next;
00083
00084
       temp->next = new_node;
00085
       pthread_mutex_unlock (&list_mutex);
00086
       return 0;
00087 }
```

7.11.2 Variable Documentation

7.11.2.1 list_mutex

```
pthread_mutex_t list_mutex = PTHREAD_MUTEX_INITIALIZER
```

Definition at line 14 of file linked_list.c.

7.12 linked_list.c

Go to the documentation of this file.

```
00001 #include "linked_list.h"
00003 #include <pthread.h>
00004 #include <stdlib.h>
00005
00006 struct Node
00007 {
00008 void *ptr;
00009
       size_t size;
00010 struct Node **children;
00011
       struct Node *next;
00012 };
00013
00014 pthread_mutex_t list_mutex = PTHREAD_MUTEX_INITIALIZER;
00015
00016 struct Node *
00017 create_node (void *ptr, size_t amount, void **children)
00018 {
00019
       struct Node *new_node = (struct Node *)malloc (sizeof (struct Node));
     if (!new_node)
00020
00021
         {
```

7.12 linked list.c 71

```
00022
            return NULL;
00023
00024
        new_node->ptr = ptr;
        new_node->size = amount;
00025
        new_node->children = NULL;
00026
00027
        if (amount <= 0)
00029
            new_node->next = NULL;
00030
            return new_node;
00031
00032
        new_node->children
            = (struct Node **) malloc (sizeof (struct Node *) * amount);
00033
00034
        if (!new_node->children)
00035
00036
            free (new_node);
00037
            return NULL;
00038
00039
        for (size_t i = 0; i < amount; ++i)</pre>
00040
00041
            new_node->children[i] = (struct Node *)malloc (sizeof (struct Node));
00042
00043
            if (!new_node->children[i])
00044
00045
                for (size_t j = 0; j < i; ++j)</pre>
00046
00047
                    free (new_node->children[j]);
00048
00049
                free (new_node);
00050
                return NULL;
00051
00052
00053
        for (size_t i = 0; i < amount; ++i)</pre>
00054
00055
            new_node->children[i]->size = 0;
            new_node->children[i]->next = NULL;
new_node->children[i]->ptr = children[i];
00056
00057
00058
           new_node->children[i]->children = NULL;
00059
00060
        new_node->next = NULL;
00061
        return new_node;
00062 }
00063
00064 int.
00065 list_insert (struct Node **head, void *ptr, size_t amount, void **children)
00066 {
        struct Node *new_node = create_node (ptr, amount, children);
00067
00068
        if (!new_node)
00069
00070
            return 1:
00071
00072
        pthread_mutex_lock (&list_mutex);
00073
        if (!*head)
00074
00075
            *head = new_node;
00076
            pthread_mutex_unlock (&list_mutex);
00077
            return 0;
00079
        struct Node *temp = *head;
08000
        while (temp->next)
00081
00082
            temp = temp->next;
00083
00084
        temp->next = new_node;
00085
       pthread_mutex_unlock (&list_mutex);
00086
        return 0;
00087 }
00088
00089 void
00090 free_node (struct Node *current)
00091 {
00092
        if (current->size > 0)
00093
00094
            for (size_t i = 0; i < current->size; ++i)
00095
00096
                free (current->children[i]->ptr);
00097
                free (current->children[i]);
00098
00099
            free (current->children);
00100
00101
       free (current->ptr):
00102
       free (current);
00103 }
00104
00105 void
00106 list_delete_node (struct Node **head, const void *const ptr)
00107 {
       pthread mutex lock (&list mutex);
00108
```

```
struct Node *current = *head;
00110
       struct Node *prev = NULL;
00111
00112
       if (current && current->ptr == ptr)
00113
00114
            *head = current->next;
            free_node (current);
00115
00116
           pthread_mutex_unlock (&list_mutex);
00117
00118
00119
00120
       while (current && current->ptr != ptr)
00121
00122
           prev = current;
00123
            current = current->next;
00124
00125
00126
       if (!current)
           pthread_mutex_unlock (&list_mutex);
00129
        }
00130
00131
       prev->next = current->next;
00132
00133
       free_node (current);
00134
     pthread_mutex_unlock (&list_mutex);
00135 }
00136
00137 void
00138 list_free (struct Node **head)
00139 {
00140
       pthread_mutex_lock (&list_mutex);
00141
       struct Node *current = *head;
00142
       struct Node *next;
00143
00144
       while (current)
00145
        {
           next = current->next;
00147
           free_node (current);
00148
           current = next;
00149
00150
       *head = NULL:
00151
       pthread_mutex_unlock (&list_mutex);
00152
00153 }
```

7.13 /_w/saurion/saurion/src/low_saurion.c File Reference

```
#include "low saurion.h"
#include "config.h"
#include "linked_list.h"
#include "threadpool.h"
#include <arpa/inet.h>
#include <bits/socket-constants.h>
#include <liburing.h>
#include <liburing/io_uring.h>
#include <nanologger.h>
#include <netinet/in.h>
#include <pthread.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/eventfd.h>
#include <sys/socket.h>
#include <sys/uio.h>
#include <time.h>
#include <unistd.h>
Include dependency graph for low_saurion.c:
```

Classes

- · struct request
- · struct saurion_wrapper
- · struct chunk_params

Macros

- #define EV ACC 0
- #define EV REA 1
- #define EV WRI 2
- #define EV WAI 3
- #define EV ERR 4
- #define MIN(a, b) ((a) < (b) ? (a) : (b))
- #define MAX(a, b) ((a) > (b) ? (a) : (b))

Functions

- static uint32 t next (struct saurion *s)
- static uint64 t htonll (uint64 t value)
- static uint64 t ntohll (uint64 t value)
- void free_request (struct request *req, void **children_ptr, size_t amount)
- int initialize_iovec (struct iovec *iov, size_t amount, size_t pos, const void *msg, size_t size, uint8_t h)

Initializes a specified iovec structure with a message fragment.

- int allocate iovec (struct iovec *iov, size t amount, size t pos, size t size, void **chd ptr)
- int set_request (struct request **r, struct Node **I, size_t s, const void *m, uint8_t h)

Sets up a request and allocates iovec structures for data handling in liburing.

- static void add_accept (struct saurion *const s, struct sockaddr_in *const ca, socklen_t *const cal)
- static void add_fd (struct saurion *const s, int client_socket, int sel)
- static void add_efd (struct saurion *const s, const int client_socket, int sel)
- static void add read (struct saurion *const s, const int client socket)
- static void add read continue (struct saurion *const s, struct request *oreq, const int sel)
- static void add_write (struct saurion *const s, int fd, const char *const str, const int sel)
- static void handle_accept (const struct saurion *const s, const int fd)
- static size_t calculate_max_iov_content (const struct request *req)
- static int handle_previous_message (struct chunk_params *p)
- static int handle partial message (struct chunk params *p)
- static int handle_new_message (struct chunk_params *p)
- static int prepare_destination (struct chunk_params *p)
- static void copy_data (struct chunk_params *p, uint8_t *ok)
- static uint8_t validate_and_update (struct chunk_params *p, uint8_t ok)
- static void read_chunk_free (struct chunk_params *p)
- int read_chunk (void **dest, size_t *len, struct request *const req)

Reads a message chunk from the request's iovec buffers, handling messages that may span multiple iovec entries.

- static void handle_read (struct saurion *const s, struct request *const req)
- static void handle_write (const struct saurion *const s, const int fd)
- static void handle_error (const struct saurion *const s, const struct request *const req)
- static void handle_close (const struct saurion *const s, const struct request *const req)
- int saurion_set_socket (const int p)

Creates a socket.

struct saurion * saurion_create (uint32_t n_threads)

Creates an instance of the saurion structure.

static void handle_event_read (const struct io_uring_cqe *const cqe, struct saurion *const s, struct request *req)

- static int saurion_worker_master_loop_it (struct saurion *const s, struct sockaddr_in *client_addr, socklen_t *client_addr_len)
- void saurion_worker_master (void *arg)
- static int saurion worker slave loop it (struct saurion *const s, const int sel)
- void saurion_worker_slave (void *arg)
- int saurion_start (struct saurion *const s)

Starts event processing in the saurion structure.

void saurion_stop (const struct saurion *const s)

Stops event processing in the saurion structure.

void saurion_destroy (struct saurion *const s)

Destroys the saurion structure and frees all associated resources.

• void saurion_send (struct saurion *const s, const int fd, const char *const msg)

Sends a message through a socket using io_uring.

Variables

• static struct timespec TIMEOUT_RETRY_SPEC = { 0, TIMEOUT_RETRY * 1000L }

7.13.1 Macro Definition Documentation

7.13.1.1 EV_ACC

#define EV_ACC 0

Definition at line 26 of file low_saurion.c.

7.13.1.2 EV_ERR

#define EV_ERR 4

Definition at line 30 of file low_saurion.c.

7.13.1.3 EV_REA

#define EV REA 1

Definition at line 27 of file low saurion.c.

7.13.1.4 EV_WAI

```
#define EV_WAI 3
```

Definition at line 29 of file low_saurion.c.

7.13.1.5 EV_WRI

```
#define EV_WRI 2
```

Definition at line 28 of file low_saurion.c.

7.13.1.6 MAX

```
#define MAX(  a, \\ b ) \mbox{ ((a) } > \mbox{ (b) ? (a) : (b))}
```

Definition at line 46 of file low_saurion.c.

7.13.1.7 MIN

Definition at line 45 of file low_saurion.c.

7.13.2 Function Documentation

7.13.2.1 add_accept()

```
static void add_accept (
                struct saurion *const s,
                 struct sockaddr_in *const ca,
                 socklen_t *const cal ) [inline], [static]
Definition at line 257 of file low saurion.c.
00259 {
00260
         int res = ERROR_CODE;
00261
         pthread_mutex_lock (&s->m_rings[0]);
         while (res != SUCCESS_CODE)
00262
00263
             struct io_uring_sqe *sqe = io_uring_get_sqe (&s->rings[0]);
00265
             while (!sqe)
00266
               {
                 sqe = io_uring_get_sqe (&s->rings[0]);
nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00267
00268
00269
00270
             struct request *req = NULL;
00271
             if (!set_request (&req, &s->list, 0, NULL, 0))
00272
00273
                  free (sqe);
                  nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
res = ERROR_CODE;
00274
00275
00276
                  continue;
00277
00278
              req->client_socket = 0;
00279
              req->event_type = EV_ACC;
00280
              io_uring_prep_accept (sqe, s->ss, (struct sockaddr *const)ca, cal, 0);
             io_uring_sqe_set_data (sqe, req);
if (io_uring_submit (&s->rings[0]) < 0)</pre>
00281
00282
00283
00284
                  free (sqe);
                  list_delete_node (&s->list, req);
nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
res = ERROR_CODE;
00285
00286
00287
00288
                  continue;
00290
              res = SUCCESS_CODE;
00291
        pthread_mutex_unlock (&s->m_rings[0]);
00292
00293 }
```

7.13.2.2 add efd()

Definition at line 336 of file low saurion.c.

```
00337 {
00338    add_fd (s, client_socket, sel);
00339 }
```

7.13.2.3 add_fd()

Definition at line 297 of file low_saurion.c.

```
00299
        int res = ERROR_CODE;
00300
        pthread_mutex_lock (&s->m_rings[sel]);
        while (res != SUCCESS_CODE)
00301
00302
            struct io_uring *ring = &s->rings[sel];
00304
            struct io_uring_sqe *sqe = io_uring_get_sqe (ring);
00305
            while (!sqe)
00306
00307
                sqe = io_uring_get_sqe (ring);
                nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00308
00309
00310
            struct request *req = NULL;
00311
            if (!set_request (&req, &s->list, CHUNK_SZ, NULL, 0))
00312
               free (sqe);
res = ERROR_CODE;
00313
00314
00315
                continue;
00316
00317
           req->event_type = EV_REA;
00318
            req->client_socket = client_socket;
00319
            io_uring_prep_readv (sqe, client_socket, &req->iov[0], req->iovec_count,
00320
                                 0):
00321
            io_uring_sqe_set_data (sqe, req);
00322
            if (io_uring_submit (ring) < 0)
00323
                free (sqe);
00324
                list_delete_node (&s->list, req);
00325
00326
                res = ERROR_CODE;
00327
                continue:
00328
00329
            res = SUCCESS_CODE;
00330
00331
       pthread_mutex_unlock (&s->m_rings[sel]);
00332 }
```

7.13.2.4 add_read()

Definition at line 343 of file low saurion.c.

```
00344 {
00345    int sel = next (s);
00346    add_fd (s, client_socket, sel);
00347 }
```

7.13.2.5 add_read_continue()

Definition at line 351 of file low_saurion.c.

```
00353 {
00354
        pthread_mutex_lock (&s->m_rings[sel]);
int res = ERROR_CODE;
00355
        while (res != SUCCESS_CODE)
00356
00357
00358
            struct io_uring *ring = &s->rings[sel];
00359
            struct io_uring_sqe *sqe = io_uring_get_sqe (ring);
00360
            while (!sqe)
00361
              {
00362
                 sqe = io_uring_get_sqe (ring);
00363
                 nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
```

```
00365
             if (!set_request (&oreq, &s->list, oreq->prev_remain, NULL, 0))
00366
                 free (sqe);
res = ERROR_CODE;
00367
00368
00369
                 continue;
00370
00371
             io_uring_prep_readv (sqe, oreq->client_socket, &oreq->iov[0],
00372
                                    oreq->iovec_count, 0);
             io_uring_sqe_set_data (sqe, oreq);
if (io_uring_submit (ring) < 0)</pre>
00373
00374
00375
              {
00376
                  free (sge);
00377
                  list_delete_node (&s->list, oreq);
00378
                  res = ERROR_CODE;
00379
                 continue;
00380
00381
             res = SUCCESS_CODE;
00382
00383
        pthread_mutex_unlock (&s->m_rings[sel]);
00384 }
```

7.13.2.6 add_write()

```
static void add_write (
          struct saurion *const s,
          int fd,
          const char *const str,
          const int sel ) [inline], [static]
```

Definition at line 388 of file low_saurion.c.

```
00391
       int res = ERROR_CODE;
00392
       pthread_mutex_lock (&s->m_rings[sel]);
00393
        while (res != SUCCESS_CODE)
00394
00395
           struct io_uring *ring = &s->rings[sel];
00396
           struct io_uring_sqe *sqe = io_uring_get_sqe (ring);
00397
           while (!sqe)
00398
               sqe = io_uring_get_sqe (ring);
00399
00400
               nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00401
           struct request *req = NULL;
00402
00403
           if (!set_request (&req, &s->list, strlen (str), (const void *const)str,
00404
                             1))
00405
               free (sqe);
res = ERROR_CODE;
00406
00407
00408
               continue;
00409
00410
           req->event_type = EV_WRI;
00411
            req->client_socket = fd;
           00412
00413
            io_uring_sqe_set_data (sqe, req);
00414
00415
           if (io_uring_submit (ring) < 0)</pre>
00416
             {
00417
               free (sqe);
               list_delete_node (&s->list, req);
00418
               res = ERROR_CODE;
00419
               nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00420
00421
               continue;
00422
00423
           res = SUCCESS_CODE;
00424
       pthread_mutex_unlock (&s->m_rings[sel]);
00425
00426 }
```

7.13.2.7 calculate_max_iov_content()

7.13.2.8 copy_data()

Definition at line 584 of file low_saurion.c.

```
00585 {
00586
       size_t curr_iov_msg_rem = 0;
00587
       *ok = 1UL;
       while (1)
00588
00589
           curr_iov_msg_rem = MIN (
00591
              p->cont_rem, (p->req->iov[p->curr_iov].iov_len - p->curr_iov_off));
           00592
00593
00594
                   curr_iov_msg_rem);
           p->dest_off += curr_iov_msg_rem;
p->curr_iov_off += curr_iov_msg_rem;
00595
00596
00597
           p->cont_rem -= curr_iov_msg_rem;
00598
00599
           if (p->cont_rem <= 0)</pre>
00600
00601
               if (*(((uint8_t *)p->req->iov[p->curr_iov].iov_base)
00602
                     + p->curr_iov_off)
00603
                   != 0)
00604
                {
00605
                  *ok = OUL;
00606
00607
               *p->len = p->cont_sz;
00608
               ++p->curr_iov_off;
00609
               break;
00610
00611
           if (p->curr_iov_off >= (p->req->iov[p->curr_iov].iov_len))
00612
00613
               ++p->curr_iov;
00614
               if (p->curr_iov == p->req->iovec_count)
00615
00616
                   break;
00617
00618
               p->curr_iov_off = 0;
             }
00619
         }
00620
00621 }
```

7.13.2.9 handle_accept()

Definition at line 431 of file low_saurion.c.

7.13.2.10 handle_close()

Definition at line 785 of file low_saurion.c.

7.13.2.11 handle_error()

```
static void handle_error (  {\it const struct saurion *const s,} \\ {\it const struct request *const req ) } \ [inline], \ [static]
```

Definition at line 773 of file low saurion.c.

7.13.2.12 handle_event_read()

```
static void handle_event_read (
              const struct io_uring_cqe *const cqe,
              struct saurion *const s,
              struct request * req ) [inline], [static]
Definition at line 951 of file low saurion.c.
00954
        if (cqe->res < 0)
00955
00956
           handle_error (s, req);
00957
       if (cqe->res < 1)</pre>
       {
00959
00960
           handle_close (s, req);
00961
00962
       if (cqe->res > 0)
00963
       {
00964
           handle_read (s, req);
00965
00966 list_delete_node (&s->list, req);
00967 }
```

7.13.2.13 handle_new_message()

Definition at line 532 of file low_saurion.c.

```
00534
       p->curr_iov = 0;
00535
      p->curr_iov_off = 0;
00536
00538
                              + p->curr_iov_off);
00539 p->cont_sz = ntohll (p->cont_sz);
00540
      p->curr_iov_off += sizeof (uint64_t);
      p->cont_rem = p->cont_sz;
p->dest_off = p->cont_sz - p->cont_rem;
00541
00542
00543
00544
       if (p->cont_rem <= p->max_iov_cont)
00545
00546
          *p->dest = malloc (p->cont_sz);
00547
          if (!*p->dest)
00548
           {
00549
              return ERROR_CODE; // Error al asignar memoria.
00550
00551
          p->dest_ptr = *p->dest;
00552
00553
00554
00555
         p->req->prev = malloc (p->cont_sz);
00556
          if (!p->req->prev)
          {
00557
00558
              return ERROR_CODE; // Error al asignar memoria.
00559
00560
          p->dest_ptr = p->req->prev;
00561
          *p->dest = NULL;
00562
      return SUCCESS_CODE;
00563
00564 }
```

7.13.2.14 handle_partial_message()

```
static int handle_partial_message (
              struct chunk_params * p ) [inline], [static]
Definition at line 494 of file low saurion.c.
00495
00496
        p->curr_iov = p->req->next_iov;
       p->curr_iov_off = p->req->next_offset;
00498
00499
       p->cont_sz = *(size_t *)((uint8_t *)p->req->iov[p->curr_iov].iov_base
00500
                                 + p->curr_iov_off);
00501
       p->cont_sz = ntohll (p->cont_sz);
       p->curr_iov_off += sizeof (uint64_t);
p->cont_rem = p->cont_sz;
00502
00504
       p->dest_off = p->cont_sz - p->cont_rem;
00505
00506
       if ((p->curr_iov_off + p->cont_rem + 1) <= p->max_iov_cont)
00507
00508
            *p->dest = malloc (p->cont_sz);
            if (!*p->dest)
00509
00510
00511
                return ERROR_CODE;
00512
00513
           p->dest_ptr = *p->dest;
00514
00515
       else
00516
        {
00517
           p->req->prev = malloc (p->cont_sz);
00518
           if (!p->req->prev)
00519
00520
               return ERROR_CODE;
00521
           p->dest_ptr = p->req->prev;
00523
            *p->dest = NULL;
            \star p \rightarrow len = 0;
00524
00525
       return SUCCESS_CODE;
00526
00527 }
```

7.13.2.15 handle previous message()

Definition at line 469 of file low saurion.c.

```
00470 {
          p->cont_sz = p->req->prev_size;
00471
          p->cont_sz = p->req->prev_size;
p->cont_rem = p->req->prev_remain;
p->dest_off = p->cont_sz - p->cont_rem;
00472
00473
00475
          if (p->cont_rem <= p->max_iov_cont)
00476
               *p->dest = p->req->prev;
p->dest_ptr = *p->dest;
p->req->prev = NULL;
00477
00478
00479
00480
               p->req->prev_size = 0;
00481
               p->req->prev_remain = 0;
00482
00483
          else
           p->dest_ptr = p->req->prev;
*p->dest = NULL;
}
00484
          {
00485
00487
00488
          return SUCCESS_CODE;
00489 }
```

7.13.2.16 handle_read()

```
static void handle_read (
              struct saurion *const s,
              struct request *const req ) [inline], [static]
Definition at line 724 of file low_saurion.c.
00725 {
        void *msg = NULL;
00726
       size_t len = 0;
while (1)
00727
00728
00729
         {
00730
            if (!read_chunk (&msg, &len, req))
00731
              {
               break;
00732
00733
00734
            if (req->next_iov || req->next_offset)
00735
00736
                if (s->cb.on_readed && msg)
00737
00738
                   s->cb.on_readed (req->client_socket, msg, len,
00739
                                    s->cb.on_readed_arg);
00740
00741
               free (msg);
00742
               msg = NULL;
00743
                continue;
00744
00745
            if (req->prev && req->prev_size && req->prev_remain)
00746
            {
00747
               add read continue (s, reg, next (s));
00748
               return;
00749
00750
            if (s->cb.on_readed && msg)
00751
            {
00752
               s->cb.on_readed (req->client_socket, msg, len, s->cb.on_readed_arg);
00753
00754
           free (msg);
00755
           msg = NULL;
00756
           break;
00757
00758
       add_read (s, req->client_socket);
00759 }
```

7.13.2.17 handle_write()

Definition at line 763 of file low saurion.c.

7.13.2.18 htonll()

Definition at line 66 of file low saurion.c.

```
00067 {
00068 int num = 42;
```

7.13.2.19 next()

```
static uint32_t next ( struct \ saurion \ * \ s \ ) \ \ [inline], \ [static]
```

Definition at line 58 of file low saurion.c.

```
00059 {
00060    s->next = (s->next + 1) % s->n_threads;
00061    return s->next;
00062 }
```

7.13.2.20 ntohll()

Definition at line 80 of file low_saurion.c.

```
00081 {
00082    int num = 42;
00083    if (*(char *)&num == 42)
00084    {
        uint32_t high_part = ntohl ((uint32_t) (value » 32));
00086        uint32_t low_part = ntohl ((uint32_t) (value & 0xfffffffffll));
00087        return ((uint64_t)low_part « 32) | high_part;
00088    }
00089    return value;
```

7.13.2.21 prepare_destination()

Definition at line 569 of file low saurion.c.

```
00570 {
00571    if (p->req->prev && p->req->prev_size && p->req->prev_remain)
00572    {
00573         return handle_previous_message (p);
00574    }
00575    if (p->req->next_iov || p->req->next_offset)
00576    {
00577         return handle_partial_message (p);
00578    }
00579    return handle_new_message (p);
00580 }
```

7.13.2.22 read_chunk_free()

```
static void read chunk free (
              struct chunk_params * p ) [inline], [static]
Definition at line 662 of file low saurion.c.
00663
00664
        free (p->dest_ptr);
        p->dest_ptr = NULL;
00665
00666
        *p->dest = NULL;
00667
       *p->len = 0;
00668
        p->req->next_iov = 0;
        p->req->next_offset = 0;
for (size_t i = p->curr_iov; i < p->req->iovec_count; ++i)
00669
00670
00672
            for (size_t j = p->curr_iov_off; j < p->req->iov[i].iov_len; ++j)
00673
00674
                uint8_t foot = *((uint8_t *)p->req->iov[i].iov_base) + j;
00675
                if (foot == 0)
00676
                  {
00677
                   p->req->next_iov = i;
00678
                    p->req->next_offset = (j + 1) % p->req->iov[i].iov_len;
00679
                    return;
00680
                  }
00681
             }
00682
          }
00683 }
```

7.13.2.23 saurion worker master()

Definition at line 1029 of file low_saurion.c.

```
01030 {
        LOG_INIT (" ");
01031
01032
        struct saurion *const s = (struct saurion *)arg;
01033
        struct sockaddr_in client_addr;
01034
       socklen_t client_addr_len = sizeof (client_addr);
01035
01036
       add_efd (s, s->efds[0], 0);
       add_accept (s, &client_addr, &client_addr_len);
01037
01038
01039
       pthread_mutex_lock (&s->status_m);
01040
        ++s->status;
01041
        pthread_cond_broadcast (&s->status_c);
01042
        pthread_mutex_unlock (&s->status_m);
01043
        while (1)
01044
        {
01045
            int ret
01046
                = saurion_worker_master_loop_it (s, &client_addr, &client_addr_len);
01047
            if (ret == ERROR_CODE || ret == CRITICAL_CODE)
01048
             {
01049
                break;
01050
              }
01051
01052
       pthread_mutex_lock (&s->status_m);
01053
         --s->status;
01054
        pthread_cond_signal (&s->status_c);
       pthread_mutex_unlock (&s->status_m);
LOG_END (" ");
01055
01056
01057
       return;
01058 }
```

7.13.2.24 saurion_worker_master_loop_it()

```
static int saurion_worker_master_loop_it (
               struct saurion *const s,
               struct sockaddr_in * client_addr,
               socklen_t * client_addr_len ) [inline], [static]
Definition at line 972 of file low saurion.c.
00976
        LOG_INIT (" ");
00977
        struct io_uring ring = s->rings[0];
00978
        struct io_uring_cqe *cqe = NULL;
00979
        int ret = io_uring_wait_cqe (&ring, &cqe);
        if (ret < 0)
00980
00981
         {
            free (cqe);
LOG_END (" ");
00982
00983
            return CRITICAL_CODE;
00984
00985
00986
        struct request *req = (struct request *)cqe->user_data;
00987
        if (!req)
00988
         {
00989
            io_uring_cqe_seen (&s->rings[0], cqe);
LOG_END (" ");
00990
            return SUCCESS_CODE;
00991
00992
00993
        if (cqe->res < 0)
00994
            list_delete_node (&s->list, req);
LOG_END (" ");
00995
00996
            return CRITICAL_CODE;
00997
00998
00999
        if (req->client_socket == s->efds[0])
01000
         {
01001
            io_uring_cqe_seen (&s->rings[0], cqe);
            list_delete_node (&s->list, req);
LOG_END (" ");
01002
01003
01004
            return ERROR_CODE;
01005
01006
        io_uring_cge_seen (&s->rings[0], cge);
        switch (req->event_type)
01008
01009
          case EV_ACC:
           handle_accept (s, cqe->res);
add_accept (s, client_addr, client_addr_len);
01010
01011
            add_read (s, cqe->res);
01012
01013
            list_delete_node (&s->list, req);
01014
            break;
01015
         case EV_REA:
01016
            handle_event_read (cqe, s, req);
01017
            break:
          case EV_WRI:
01018
           handle_write (s, req->client_socket);
01019
01020
             list_delete_node (&s->list, req);
01021
01022
       LOG_END (" ");
01023
       return SUCCESS_CODE;
01024
01025 }
```

7.13.2.25 saurion_worker_slave()

add_efd (s, s->efds[sel], sel);

01122

```
01123
01124
        pthread_mutex_lock (&s->status_m);
         .
++s->status;
01125
        pthread_cond_broadcast (&s->status_c);
01126
        pthread_mutex_unlock (&s->status_m);
01127
01128
        while (1)
01129
01130
            int res = saurion_worker_slave_loop_it (s, sel);
01131
            if (res == ERROR_CODE || res == CRITICAL_CODE)
01132
                break:
01133
              }
01134
01135
01136
       pthread_mutex_lock (&s->status_m);
01137
         --s->status;
01138
        pthread_cond_signal (&s->status_c);
       pthread_mutex_unlock (&s->status_m);
LOG_END (" ");
01139
01140
01141
        return;
01142 }
```

7.13.2.26 saurion worker slave loop it()

Definition at line 1063 of file low_saurion.c.

```
01065
        LOG_INIT (" ");
        struct io_uring ring = s->rings[sel];
01066
01067
        struct io_uring_cqe *cqe = NULL;
01068
01069
        add_efd (s, s->efds[sel], sel);
01070
        int ret = io_uring_wait_cqe (&ring, &cqe);
01071
        if (ret < 0)
01072
         {
01073
            free (cqe);
LOG_END (" ");
01074
01075
            return CRITICAL_CODE;
01076
01077
        struct request *req = (struct request *)cqe->user_data;
01078
        if (!req)
01079
            io_uring_cqe_seen (&ring, cqe);
LOG_END (" ");
01080
01081
            return SUCCESS_CODE;
01082
01083
01084
        if (cqe->res < 0)</pre>
01085
         {
            list_delete_node (&s->list, req);
LOG_END (" ");
01086
01087
            return CRITICAL_CODE;
01089
01090
        if (req->client_socket == s->efds[sel])
01091
01092
            io_uring_cqe_seen (&ring, cqe);
            list_delete_node (&s->list, req);
LOG_END (" ");
01093
01094
01095
            return ERROR_CODE;
01096
01097
        io_uring_cqe_seen (&ring, cqe);
01098
        switch (req->event_type)
01099
         {
case EV_REA:
01100
           handle_event_read (cqe, s, req);
01101
01102
            break;
01103
           case EV_WRI:
            handle_write (s, req->client_socket);
01104
01105
             list_delete_node (&s->list, req);
01106
            break;
01107
01108
        LOG_END (" ");
       return SUCCESS_CODE;
01109
01110 }
```

7.13.2.27 validate_and_update()

```
static uint8_t validate_and_update (
               struct chunk_params * p,
               uint8_t ok ) [inline], [static]
Definition at line 626 of file low_saurion.c.
00628
        if (p->req->prev)
00629
            p->req->prev_size = p->cont_sz;
p->req->prev_remain = p->cont_rem;
00630
00631
00632
            *p->dest = NULL;
            \star p \rightarrow len = 0;
00633
00634
00635
        else
00636
        {
00637
            p->req->prev size = 0;
            p->req->prev_remain = 0;
00639
00640
        if (p->curr_iov < p->req->iovec_count)
00641
        {
            uint64_t next_sz
00642
00643
                = *(uint64_t *)(((uint8_t *)p->req->iov[p->curr_iov].iov_base)
                                  + p->curr_iov_off);
00644
00645
            if ((p->req->iov[p->curr_iov].iov_len > p->curr_iov_off) && next_sz)
00646
                 p->req->next_iov = p->curr_iov;
p->req->next_offset = p->curr_iov_off;
00647
00648
00649
00650
            else
00651
            {
00652
                p->req->next_iov = 0;
00653
                p->req->next_offset = 0;
00654
00655
        }
00656
        return ok ? SUCCESS_CODE : ERROR_CODE;
```

7.13.3 Variable Documentation

7.13.3.1 TIMEOUT_RETRY_SPEC

```
struct timespec TIMEOUT_RETRY_SPEC = { 0, TIMEOUT_RETRY * 1000L } [static]
```

Definition at line 48 of file low saurion.c.

7.14 low_saurion.c

Go to the documentation of this file.

7.14 low saurion.c 89

```
00015 #include <stdlib.h>
                                       // for free, malloc
00016 #include <string.h>
                                      // for memset, memcpy, strlen
                                      // for eventfd, EFD_NONBLOCK
00017 #include <sys/eventfd.h>
                                       // for socklen_t, bind, listen, setsockopt
00018 #include <sys/socket.h>
00019 #include <sys/uio.h>
                                      // for iovec
00020 #include <time.h>
                                       // for nanosleep
00021 #include <unistd.h>
                                       // for close, write
00022
00023 struct Node;
00024 struct iovec;
00025
00026 #define EV_ACC 0
00027 #define EV_REA 1
00028 #define EV_WRI 2
00029 #define EV_WAI 3
00030 #define EV_ERR 4
00031
00032 struct request
00033 {
00034
       void *prev;
00035
       size_t prev_size;
00036
       size_t prev_remain;
00037
       size_t next_iov;
00038
       size t next offset;
00039
       int event_type;
00040 size_t iovec_count;
00041
        int client_socket;
00042 struct iovec iov[];
00043 };
00044
00045 #define MIN(a, b) ((a) < (b) ? (a) : (b))
00046 #define MAX(a, b) ((a) > (b) ? (a) : (b))
00047
00048 static struct timespec TIMEOUT_RETRY_SPEC = { 0, TIMEOUT_RETRY * 1000L };
00049
00050 struct saurion_wrapper
00051 {
       struct saurion *s;
00053
        uint32_t sel;
00054 };
00055
00056 // next
00057 static inline uint32 t
00058 next (struct saurion *s)
00059 {
00060 s->next = (s->next + 1) % s->n_threads;
00061 return s->next;
00062 }
00063
00064 // htonll
00065 static inline uint64_t
00066 htonll (uint64_t value)
00067 {
00068
        int num = 42;
00069
        if (*(char *)&num == 42)
        {
00070
00071
            uint32_t high_part = hton1 ((uint32_t) (value » 32));
00072
            uint32_t low_part = hton1 ((uint32_t) (value & 0xFFFFFFFFLL));
           return ((uint64_t)low_part « 32) | high_part;
00073
        }
00074
00075
       return value;
00076 }
00077
00078 // ntohll
00079 static inline uint64_t
00080 ntohll (uint64_t value)
00081 {
00082
        int num = 42;
00083
        if (*(char *)&num == 42)
        {
            uint32_t high_part = ntohl ((uint32_t)(value » 32));
uint32_t low_part = ntohl ((uint32_t)(value & 0xFFFFFFFLL));
00085
00086
00087
           return ((uint64_t)low_part « 32) | high_part;
00088
00089
       return value;
00090 }
00091
00092 // free_request
00093 void
00094 free request (struct request *req, void **children ptr, size t amount)
00095 {
00096
        if (children_ptr)
00097
00098
            free (children_ptr);
00099
           children_ptr = NULL;
00100
00101
       for (size_t i = 0; i < amount; ++i)</pre>
```

```
{
00103
          free (req->iov[i].iov_base);
00104
            req->iov[i].iov_base = NULL;
00105
00106
        free (reg);
        req = NULL;
00107
        free (children_ptr);
00109
        children_ptr = NULL;
00110 }
00111
00112 // initialize iovec
00113 [[nodiscard]]
00114 int
00115 initialize_iovec (struct iovec *iov, size_t amount, size_t pos,
00116
                         const void *msg, size_t size, uint8_t h)
00117 {
        if (!iov || !iov->iov_base)
00118
00119
        {
            return ERROR_CODE;
00121
00122
        if (msg)
00123
            size_t len = iov->iov_len;
char *dest = (char *)iov->iov_base;
char *orig = (char *)msg + pos * CHUNK_SZ;
00124
00125
00126
00127
            size_t cpy_sz = 0;
00128
             if (h)
00129
                 if (pos == 0)
00130
00131
                  {
                    uint64_t send_size = htonll (size);
memcpy (dest, &send_size, sizeof (uint64_t));
dest += sizeof (uint64_t);
00132
00133
00134
00135
                     len -= sizeof (uint64_t);
00136
00137
                 else
00138
                  {
00139
                     orig -= sizeof (uint64_t);
00140
00141
                 if ((pos + 1) == amount)
00142
                     --len;
00143
                     cpy_sz = (len < size ? len : size);
00144
00145
                     dest[cpy_sz] = 0;
00146
00147
              }
00148
            cpy_sz = (len < size ? len : size);</pre>
00149
            memcpy (dest, orig, cpy_sz);
            dest += cpy_sz;
size_t rem = CHUNK_SZ - (dest - (char *)iov->iov_base);
00150
00151
00152
            memset (dest, 0, rem);
00153
00154
        else
00155
        {
            memset ((char *)iov->iov_base, 0, CHUNK_SZ);
00156
00157
        return SUCCESS_CODE;
00159 }
00160
00161 // allocate iovec
00162 [[nodiscard]]
00163 int
00164 allocate_iovec (struct iovec *iov, size_t amount, size_t pos, size_t size,
00165
                       void **chd_ptr)
00166 {
00167
        if (!iov || !chd_ptr)
00168
            return ERROR_CODE;
00169
00170
        iov->iov_base = malloc (CHUNK_SZ);
00172
        if (!iov->iov_base)
00173
00174
            return ERROR_CODE;
00175
00176
        iov->iov_len = (pos == (amount - 1) ? (size % CHUNK_SZ) : CHUNK_SZ);
00177
        if (iov->iov_len == 0)
00178
        {
00179
            iov->iov_len = CHUNK_SZ;
00180
00181
        chd ptr[pos] = iov->iov base;
        return SUCCESS_CODE;
00182
00183 }
00184
00185 // set_request
00186 [[nodiscard]]
00187 int.
00188 set request (struct request **r, struct Node **l, size t s, const void *m,
```

7.14 low_saurion.c 91

```
00189
                   uint8_t h)
00190 {
00191
        uint64_t full_size = s;
00192
        if (h)
00193
            full_size += (sizeof (uint64_t) + sizeof (uint8_t));
00194
00195
00196
        size_t amount = full_size / CHUNK_SZ;
00197
        amount = amount + (full_size % CHUNK_SZ == 0 ? 0 : 1);
00198
        struct request *temp = (struct request *)malloc (
            sizeof (struct request) + sizeof (struct iovec) * amount);
00199
00200
        if (!temp)
        {
00201
00202
            return ERROR_CODE;
00203
00204
        if (!*r)
00205
00206
            *r = temp;
00207
            (*r) \rightarrow prev = NULL;
00208
            (*r) \rightarrow prev\_size = 0;
00209
            (*r)->prev_remain = 0;
00210
            (*r) \rightarrow next_iov = 0;
00211
            (*r)->next_offset = 0;
00212
00213
        else
00214
         {
00215
            temp->client_socket = (*r)->client_socket;
00216
            temp->event_type = (*r)->event_type;
00217
            temp->prev = (*r)->prev;
            temp->prev_size = (*r)->prev_size;
00218
            temp->prev_remain = (*r)->prev_remain;
00219
00220
            temp->next_iov = (*r)->next_iov;
00221
            temp->next_offset = (*r)->next_offset;
00222
            *r = temp;
00223
00224
        struct request *req = *r;
        req->iovec_count = (int)amount;
00225
        void **children_ptr = (void **)malloc (amount * sizeof (void *));
00227
        if (!children_ptr)
00228
00229
            free_request (req, children_ptr, 0);
            return ERROR CODE;
00230
00231
00232
        for (size_t i = 0; i < amount; ++i)</pre>
00233
00234
            if (!allocate_iovec (&req->iov[i], amount, i, full_size, children_ptr))
00235
00236
                free_request (req, children_ptr, amount);
00237
                return ERROR CODE:
00238
00239
            if (!initialize_iovec (&req->iov[i], amount, i, m, s, h))
00240
00241
                free_request (req, children_ptr, amount);
00242
                return ERROR_CODE;
00243
00244
00245
        if (list_insert (l, req, amount, children_ptr))
00246
        {
00247
            free_request (req, children_ptr, amount);
00248
            return ERROR_CODE;
          }
00249
       free (children_ptr);
00250
00251
       return SUCCESS_CODE;
00252 }
00253
00254 /************ ADDERS **************/
00255 // add_accept
00256 static inline void
00257 add_accept (struct saurion *const s, struct sockaddr_in *const ca,
00258
                  socklen_t *const cal)
00259 {
00260
       int res = ERROR_CODE;
00261
        pthread_mutex_lock (&s->m_rings[0]);
00262
        while (res != SUCCESS_CODE)
00263
00264
            struct io_uring_sqe *sqe = io_uring_get_sqe (&s->rings[0]);
00265
            while (!sqe)
00266
                sqe = io_uring_get_sqe (&s->rings[0]);
nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00267
00268
00269
00270
            struct request *req = NULL;
00271
            if (!set_request (&req, &s->list, 0, NULL, 0))
00272
00273
                free (sqe);
                nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
res = ERROR_CODE;
00274
00275
```

```
continue;
00277
00278
            req->client_socket = 0;
00279
            req->event_type = EV_ACC;
            io_uring_prep_accept (sqe, s->ss, (struct sockaddr *const)ca, cal, 0);
00280
00281
            io_uring_sqe_set_data (sqe, req);
            if (io_uring_submit (&s->rings[0]) < 0)</pre>
00283
00284
                free (sqe);
                list_delete_node (&s->list, req);
nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00285
00286
                res = ERROR_CODE;
00287
00288
                continue;
00289
00290
            res = SUCCESS_CODE;
00291
00292
       pthread_mutex_unlock (&s->m_rings[0]);
00293 }
00295 // add_fd
00296 static inline void
00297 add_fd (struct saurion *const s, int client_socket, int sel)
00298 {
00299
        int res = ERROR CODE;
        pthread_mutex_lock (&s->m_rings[sel]);
00300
        while (res != SUCCESS_CODE)
00302
00303
            struct io_uring *ring = &s->rings[sel];
00304
            struct io_uring_sqe *sqe = io_uring_get_sqe (ring);
00305
            while (!sqe)
00306
             {
00307
                sqe = io_uring_get_sqe (ring);
00308
                nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00309
00310
            struct request *req = NULL;
            if (!set_request (&req, &s->list, CHUNK_SZ, NULL, 0))
00311
00312
             {
                free (sqe);
00314
                res = ERROR_CODE;
00315
                continue;
00316
00317
            req->event_type = EV_REA;
            reg->client socket = client socket;
00318
            io_uring_prep_readv (sqe, client_socket, &req->iov[0], req->iovec_count,
00319
00320
                                 0);
00321
            io_uring_sqe_set_data (sqe, req);
00322
            if (io_uring_submit (ring) < 0)</pre>
00323
              {
00324
                free (sge);
00325
                list_delete_node (&s->list, req);
                res = ERROR_CODE;
00326
00327
                continue;
00328
00329
            res = SUCCESS CODE;
00330
00331
       pthread_mutex_unlock (&s->m_rings[sel]);
00333
00334 // add_efd
00335 static inline void
00336 add_efd (struct saurion *const s, const int client_socket, int sel)
00337 {
00338
       add_fd (s, client_socket, sel);
00339 }
00340
00341 // add_read
00342 static inline void
00343 add_read (struct saurion *const s, const int client_socket)
00344 {
00345
       int sel = next (s);
00346 add_fd (s, client_socket, sel);
00347 }
00348
00349 // add_read_continue
00350 static inline void
00351 add_read_continue (struct saurion *const s, struct request *oreq,
00352
                         const int sel)
00353 {
00354
       pthread_mutex_lock (&s->m_rings[sel]);
        int res = ERROR_CODE;
00355
        while (res != SUCCESS_CODE)
00356
00357
         {
00358
            struct io_uring *ring = &s->rings[sel];
00359
            struct io_uring_sqe *sqe = io_uring_get_sqe (ring);
00360
            while (!sqe)
00361
              {
00362
                sae = io urina aet sae (rina);
```

7.14 low_saurion.c 93

```
nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00364
00365
            if (!set_request (&oreq, &s->list, oreq->prev_remain, NULL, 0))
00366
             {
00367
                free (sqe);
res = ERROR_CODE;
00368
                continue;
00369
00370
00371
            io_uring_prep_readv (sqe, oreq->client_socket, &oreq->iov[0],
00372
                                 oreq->iovec_count, 0);
            io_uring_sqe_set_data (sqe, oreq);
00373
00374
            if (io_uring_submit (ring) < 0)
00375
              {
00376
                free (sqe);
00377
                list_delete_node (&s->list, oreq);
00378
                res = ERROR_CODE;
00379
                continue:
00380
00381
            res = SUCCESS_CODE;
00382
00383
       pthread_mutex_unlock (&s->m_rings[sel]);
00384 }
00385
00386 // add_write
00387 static inline void
00388 add_write (struct saurion *const s, int fd, const char *const str,
00389
                 const int sel)
00390 {
       int res = ERROR_CODE;
00391
00392
       pthread_mutex_lock (&s->m_rings[sel]);
while (res != SUCCESS_CODE)
00393
00394
         {
00395
            struct io_uring *ring = &s->rings[sel];
00396
            struct io_uring_sqe *sqe = io_uring_get_sqe (ring);
00397
            while (!sqe)
00398
              {
                sqe = io_uring_get_sqe (ring);
nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00399
00400
00401
00402
            struct request *req = NULL;
            if (!set_request (&req, &s->list, strlen (str), (const void *const)str,
00403
00404
                              1))
00405
              {
00406
                free (sqe);
                res = ERROR_CODE;
00407
00408
                continue;
00409
00410
            req->event_type = EV_WRI;
            req->client_socket = fd;
00411
00412
            io_uring_prep_writev (sqe, req->client_socket, req->iov,
00413
                                  req->iovec_count, 0);
00414
            io_uring_sqe_set_data (sqe, req);
00415
            if (io_uring_submit (ring) < 0)</pre>
00416
             {
                free (sqe);
00417
                list_delete_node (&s->list, req);
00418
                res = ERROR_CODE;
00420
                nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00421
00422
            res = SUCCESS_CODE;
00423
00424
00425
       pthread_mutex_unlock (&s->m_rings[sel]);
00426 }
00427
00429 // handle_accept
00430 static inline void
00431 handle accept (const struct saurion *const s, const int fd)
00432 {
00433
       if (s->cb.on_connected)
        {
00434
00435
           s->cb.on_connected (fd, s->cb.on_connected_arg);
00436
00437 }
00438
00439 // calculate_max_iov_content
00440 [[nodiscard]]
00441 static inline size_t
00442 calculate max iov content (const struct request *reg)
00443 {
00444
        size_t max_iov_cont = 0;
00445
        for (size_t i = 0; i < req->iovec_count; ++i)
00446
00447
            max_iov_cont += req->iov[i].iov_len;
00448
00449
       return max iov cont;
```

```
00451
00452 struct chunk_params
00453 {
       void **dest;
00454
00455
       void *dest_ptr;
       size_t dest_off;
00456
00457
       struct request *req;
00458
       size_t cont_sz;
00459
       size_t cont_rem;
00460
       size_t max_iov_cont;
00461
       size_t curr_iov;
       size_t curr_iov_off;
00462
00463 size_t *len;
00464 };
00465
00466 // handle_previous_message
00467 [[nodiscard]]
00468 static inline int
00469 handle_previous_message (struct chunk_params *p)
00470 {
00471
       p->cont_sz = p->req->prev_size;
        p->cont_rem = p->req->prev_remain;
p->dest_off = p->cont_sz - p->cont_rem;
00472
00473
00474
00475
        if (p->cont_rem <= p->max_iov_cont)
        {
00476
            *p->dest = p->req->prev;
00477
            p->dest_ptr = *p->dest;
00478
            p->req->prev = NULL;
00479
00480
            p->req->prev_size = 0;
00481
            p->req->prev_remain = 0;
00482
00483
       else
00484
        {
         p->dest_ptr = p->req->prev;
*p->dest = NULL;
}
00485
00486
00488
       return SUCCESS_CODE;
00489 }
00490
00491 // handle_partial_message
00492 [[nodiscard]]
00493 static inline int
00494 handle_partial_message (struct chunk_params *p)
00495 {
00496 p->curr_iov = p->req->next_iov;
00497
       p->curr_iov_off = p->req->next_offset;
00498
       p->cont_sz = *(size_t *)((uint8_t *)p->req->iov[p->curr_iov].iov_base
00499
00500
                                  + p->curr_iov_off);
00501
        p->cont_sz = ntohl1 (p->cont_sz);
00502
        p->curr_iov_off += sizeof (uint64_t);
        p->cont_rem = p->cont_sz;
p->dest_off = p->cont_sz - p->cont_rem;
00503
00504
00505
00506
        if ((p->curr_iov_off + p->cont_rem + 1) <= p->max_iov_cont)
00507
         {
00508
            *p->dest = malloc (p->cont_sz);
00509
            if (!*p->dest)
00510
             {
                return ERROR_CODE;
00511
00512
00513
            p->dest_ptr = *p->dest;
00514
00515
        else
00516
         {
            p->req->prev = malloc (p->cont_sz);
00517
00518
            if (!p->req->prev)
            {
00520
               return ERROR_CODE;
00521
            p->dest_ptr = p->req->prev;
*p->dest = NULL;
00522
00523
            *p->len = 0;
00524
00525
00526
       return SUCCESS_CODE;
00527 }
00528
00529 // handle_new_message
00530 [[nodiscard]]
00531 static inline int
00532 handle_new_message (struct chunk_params *p)
00533 {
00534 p->curr_iov = 0;
       p->curr_iov_off = 0;
00535
00536
```

7.14 low_saurion.c 95

```
p->cont_sz = *(size_t *)((uint8_t *)p->req->iov[p->curr_iov].iov_base
00538
                                + p->curr_iov_off);
00539
       p->cont_sz = ntohll (p->cont_sz);
00540
       p->curr_iov_off += sizeof (uint64_t);
00541
       p->cont_rem = p->cont_sz;
00542
       p->dest_off = p->cont_sz - p->cont_rem;
00543
00544
        if (p->cont_rem <= p->max_iov_cont)
00545
00546
            *p->dest = malloc (p->cont_sz);
00547
            if (!*p->dest)
00548
             {
00549
               return ERROR_CODE; // Error al asignar memoria.
00550
00551
           p->dest_ptr = *p->dest;
00552
00553
       else
00554
         {
           p->req->prev = malloc (p->cont_sz);
00556
           if (!p->req->prev)
00557
00558
               return ERROR_CODE; // Error al asignar memoria.
00559
           p->dest_ptr = p->req->prev;
00560
00561
           *p->dest = NULL;
00562
00563
       return SUCCESS_CODE;
00564 }
00565
00566 // prepare_destination
00567 [[nodiscard]]
00568 static inline int
00569 prepare_destination (struct chunk_params *p)
00570 {
00571
           (p->req->prev && p->req->prev_size && p->req->prev_remain)
00572
00573
           return handle previous message (p);
00575
       if (p->req->next_iov || p->req->next_offset)
00576
00577
           return handle_partial_message (p);
00578
00579
       return handle new message (p);
00580 }
00581
00582 // copy_data
00583 static inline void
00584 copy_data (struct chunk_params *p, uint8_t *ok)
00585 {
00586
       size_t curr_iov_msg_rem = 0;
00587
       *ok = 1UL;
00588
       while (1)
00589
00590
           curr_{iov_msg_rem} = MIN (
               p->cont_rem, (p->req->iov[p->curr_iov].iov_len - p->curr_iov_off));
00591
           00592
00594
                   curr_iov_msg_rem);
00595
           p->dest_off += curr_iov_msg_rem;
           p->curr_iov_off += curr_iov_msg_rem;
00596
00597
           p->cont_rem -= curr_iov_msg_rem;
00598
00599
            if (p->cont_rem <= 0)</pre>
00600
00601
               if (*(((uint8_t *)p->req->iov[p->curr_iov].iov_base)
                     + p->curr_iov_off)
00602
                   != 0)
00603
00604
                 {
00605
                   *ok = OUL;
00606
00607
               *p->len = p->cont_sz;
00608
               ++p->curr_iov_off;
00609
               break;
00610
              (p->curr_iov_off >= (p->req->iov[p->curr_iov].iov_len))
00611
00612
00613
00614
                if (p->curr_iov == p->req->iovec_count)
00615
00616
                   break:
00617
00618
               p->curr_iov_off = 0;
00619
00620
         }
00621 }
00622
00623 // validate and update
```

```
00624 [[nodiscard]]
00625 static inline uint8_t
00626 validate_and_update (struct chunk_params *p, uint8_t ok)
00627 {
00628
        if (p->req->prev)
00629
         {
00630
            p->req->prev_size = p->cont_sz;
00631
            p->req->prev_remain = p->cont_rem;
            *p->dest = NULL;
*p->len = 0;
00632
00633
          }
00634
00635
       else
00636
        {
00637
           p->req->prev_size = 0;
00638
            p->req->prev_remain = 0;
00639
        if (p->curr_iov < p->req->iovec_count)
00640
00641
        {
00642
            uint64_t next_sz
00643
               = *(uint64_t *)(((uint8_t *)p->req->iov[p->curr_iov].iov_base)
00644
                                 + p->curr_iov_off);
00645
            if ((p->req->iov[p->curr_iov].iov_len > p->curr_iov_off) && next_sz)
             {
00646
00647
                p->req->next_iov = p->curr_iov;
00648
                p->req->next_offset = p->curr_iov_off;
00649
00650
            else
00651
             {
00652
                p->req->next_iov = 0;
               p->req->next_offset = 0;
00653
00654
00655
         }
00656
00657
       return ok ? SUCCESS_CODE : ERROR_CODE;
00658 }
00659
00660 // read_chunk_free
00661 static inline void
00662 read_chunk_free (struct chunk_params *p)
00663 {
00664
       free (p->dest_ptr);
       p->dest_ptr = NULL;
00665
       *p->dest = NULL:
00666
       *p->len = 0;
00667
        p->req->next_iov = 0;
00668
00669
        p->req->next_offset = 0;
00670
        for (size_t i = p->curr_iov; i < p->req->iovec_count; ++i)
00671
00672
            for (size_t j = p->curr_iov_off; j < p->req->iov[i].iov_len; ++j)
00673
00674
                uint8_t foot = *((uint8_t *)p->req->iov[i].iov_base) + j;
00675
                <u>if</u> (foot == 0)
00676
00677
                    p->req->next_iov = i;
00678
                    p->req->next_offset = (j + 1) % p->req->iov[i].iov_len;
00679
                    return;
00681
              }
00682
          }
00683 }
00684
00685 // read_chunk
00686 [[nodiscard]]
00687 int
00688 read_chunk (void **dest, size_t *len, struct request *const req)
00689 {
00690
       struct chunk_params p;
00691
       p.req = req;
p.dest = dest;
00692
        p.len = len;
00693
00694
        if (p.req->iovec_count == 0)
00695
00696
            return ERROR_CODE;
00697
00698
00699
        p.max_iov_cont = calculate_max_iov_content (p.req);
00700
        p.cont_sz = 0;
00701
        p.cont_rem = 0;
        p.curr_iov = 0;
00702
        p.curr_iov_off = 0;
p.dest_off = 0;
p.dest_ptr = NULL;
00703
00704
00705
00706
        if (!prepare_destination (&p))
00707
00708
            return ERROR_CODE;
00709
00710
```

7.14 low_saurion.c 97

```
00711
       uint8_t ok = 1UL;
00712
       copy_data (&p, &ok);
00713
00714
       if (validate_and_update (&p, ok))
00715
00716
           return SUCCESS_CODE;
00717
00718
       read_chunk_free (&p);
00719
       return ERROR_CODE;
00720 }
00721
00722 // handle read
00723 static inline void
00724 handle_read (struct saurion *const s, struct request *const req)
00725 {
00726
       void *msg = NULL;
       size_t len = 0;
while (1)
00727
00728
00729
        {
00730
           if (!read_chunk (&msg, &len, req))
00731
             {
00732
               break;
00733
00734
           if (req->next_iov || req->next_offset)
00735
             {
00736
               if (s->cb.on_readed && msg)
00737
00738
                   s->cb.on_readed (req->client_socket, msg, len,
00739
                                    s->cb.on_readed_arg);
00740
                 }
00741
               free (msq);
00742
               msg = NULL;
00743
               continue;
00744
00745
           if (req->prev && req->prev_size && req->prev_remain)
00746
00747
               add_read_continue (s, req, next (s));
00748
               return;
00749
00750
           if (s->cb.on_readed && msg)
00751
               s->cb.on_readed (req->client_socket, msg, len, s->cb.on_readed_arg);
00752
00753
00754
           free (msg);
00755
           msg = NULL;
00756
           break;
00757
00758
       add_read (s, req->client_socket);
00759 }
00760
00761 // handle_write
00762 static inline void
00763 handle_write (const struct saurion *const s, const int fd)
00764 {
00765
       if (s->cb.on_wrote)
00766
        {
00767
           s->cb.on_wrote (fd, s->cb.on_wrote_arg);
00768
00769 }
00770
00771 // handle_error
00772 static inline void
00773 handle_error (const struct saurion *const s, const struct request *const req)
00774 {
00775
       if (s->cb.on_error)
00776
           const char *resp = "ERROR";
00777
           s->cb.on_error (req->client_socket, resp, (ssize_t)strlen (resp),
00778
00779
                           s->cb.on error arg);
00780
         }
00781 }
00782
00783 // handle_close
00784 static inline void
00785 handle_close (const struct saurion *const s, const struct request *const req)
00786 {
00787
       if (s->cb.on_closed)
00788
       {
00789
           s->cb.on_closed (req->client_socket, s->cb.on_closed_arg);
00790
00791
       close (req->client_socket);
00792 }
00793
00795 // saurion_set_socket
00796 [[nodiscard]] int
00797 saurion_set_socket (const int p)
```

```
00798 {
00799
        int sock = 0;
00800
        struct sockaddr_in srv_addr;
00801
00802
        sock = socket (PF_INET, SOCK_STREAM, 0);
00803
        if (sock < 1)
00805
             return ERROR_CODE;
00806
00807
00808
        int enable = 1:
        if (setsockopt (sock, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof (int)) < 0)</pre>
00809
00810
         {
00811
             return ERROR_CODE;
00812
00813
        memset (&srv_addr, 0, sizeof (srv_addr));
srv_addr.sin_family = AF_INET;
srv_addr.sin_port = htons (p);
00814
00815
00816
00817
        srv_addr.sin_addr.s_addr = htonl (INADDR_ANY);
00818
00819
        if (bind (sock, (const struct sockaddr \star)&srv_addr, sizeof (srv_addr)) < 0)
        {
00820
            return ERROR CODE:
00821
00822
00824
        if (listen (sock, ACCEPT_QUEUE) < 0)</pre>
00825
            return ERROR_CODE;
00826
00827
00828
00829
        return sock;
00830 }
00831
00832 // saurion_create
00833 [[nodiscard]]
00834 struct saurion
00835 saurion_create (uint32_t n_threads)
00836 {
00837
        LOG_INIT (" ");
00838
        struct saurion *p = (struct saurion *) malloc (size of (struct saurion));
00839
        if (!p)
00840
            LOG_END (" ");
00841
          return NULL;
00842
00843
00844
        int ret = 0;
00845
        ret = pthread_mutex_init (&p->status_m, NULL);
00846
        if (ret)
00847
         {
            free (p);
LOG_END (" ");
00848
00849
00850
             return NULL;
00851
00852
        ret = pthread_cond_init (&p->status_c, NULL);
00853
        if (ret)
00854
             free (p);
LOG_END (" ");
00855
00856
00857
             return NULL;
00858
00859
        p->m_rings
00860
             = (pthread_mutex_t *)malloc (n_threads * sizeof (pthread_mutex_t));
00861
        if (!p->m_rings)
00862
         {
            free (p);
LOG_END (" ");
00863
00864
            return NULL;
00865
00866
00867
        for (uint32_t i = 0; i < n_threads; ++i)</pre>
00868
00869
            pthread_mutex_init (&(p->m_rings[i]), NULL);
00870
00871
        p->ss = 0;
        n_threads = (n_threads < 2 ? 2 : n_threads);
n_threads = (n_threads > NUM_CORES ? NUM_CORES : n_threads);
00872
00873
00874
        p->n_threads = n_threads;
00875
        p->status = 0;
00876
        p->list = NULL;
00877
        p->cb.on_connected = NULL;
00878
        p->cb.on_connected_arg = NULL;
        p->cb.on_readed = NULL;
00879
00880
        p->cb.on_readed_arg = NULL;
00881
        p->cb.on_wrote = NULL;
00882
        p->cb.on_wrote_arg = NULL;
00883
        p->cb.on_closed = NULL;
00884
       p->cb.on_closed_arg = NULL;
```

7.14 low_saurion.c 99

```
p->cb.on_error = NULL;
00886
        p->cb.on_error_arg = NULL;
        p->next = 0;
p->efds = (int *)malloc (sizeof (int) * p->n_threads);
00887
00888
00889
        if (!p->efds)
00890
00891
            free (p->m_rings);
            free (p);
LOG_END (" ");
00892
00893
00894
            return NULL;
00895
00896
        for (uint32 t i = 0; i  n threads; ++i)
00897
00898
            p->efds[i] = eventfd (0, EFD_NONBLOCK);
00899
             if (p->efds[i] == ERROR_CODE)
00900
00901
                for (uint32_t j = 0; j < i; ++j)
00902
                 {
00903
                    close (p->efds[j]);
00904
00905
                 free (p->efds);
00906
                 free (p->m_rings);
00907
                free (p);
LOG_END (" ");
00908
00909
                 return NULL;
00910
00911
00912
        p->rings
00913
            = (struct io_uring *)malloc (sizeof (struct io_uring) * p->n_threads);
        if (!p->rings)
00914
00915
00916
            for (uint32_t j = 0; j < p->n_threads; ++j)
00917
00918
                close (p->efds[j]);
00919
00920
            free (p->efds);
00921
            free (p->m_rings);
            free (p);
LOG_END (" ");
00923
00924
            return NULL;
00925
00926
        for (uint32_t i = 0; i < p->n_threads; ++i)
00927
00928
            memset (&p->rings[i], 0, sizeof (struct io_uring));
00929
            ret = io_uring_queue_init (SAURION_RING_SIZE, &p->rings[i], 0);
00930
             if (ret)
00931
                for (uint32_t j = 0; j < p->n_threads; ++j)
00932
00933
                  {
00934
                    close (p->efds[j]);
00935
00936
                 free (p->efds);
00937
                 free (p->rings);
                 free (p->m_rings);
00938
00939
                free (p);
LOG_END (" ");
00940
00941
                 return NULL;
00942
00943
        p->pool = threadpool_create (p->n_threads);
LOG_END (" ");
00944
00945
00946
        return p;
00947 }
00948
00949 // handle_event_read
00950 static inline void
00951 handle_event_read (const struct io_uring_cqe *const cqe,
00952
                          struct saurion *const s, struct request *req)
00953 {
00954
        if (cqe->res < 0)</pre>
00955
00956
            handle_error (s, req);
00957
00958
        if (cge->res < 1)</pre>
00959
        {
00960
            handle_close (s, req);
00961
00962
        if (cqe->res > 0)
00963
00964
            handle_read (s, req);
00965
00966
        list_delete_node (&s->list, req);
00967 }
00968
00969 // saurion_worker_master_loop_it
00970 [[nodiscard]]
00971 static inline int
```

```
00972 saurion_worker_master_loop_it (struct saurion *const s,
00973
                                        struct sockaddr_in *client_addr,
00974
                                        socklen_t *client_addr_len)
00975 {
        LOG_INIT (" ");
00976
        struct io_uring ring = s->rings[0];
struct io_uring_cqe *cqe = NULL;
00977
00979
        int ret = io_uring_wait_cqe (&ring, &cqe);
00980
        if (ret < 0)
00981
            free (cqe);
LOG_END (" ");
00982
00983
00984
            return CRITICAL_CODE;
00985
00986
        struct request *req = (struct request *)cqe->user_data;
00987
        if (!req)
00988
00989
            io_uring_cqe_seen (&s->rings[0], cqe);
LOG_END (" ");
00990
00991
            return SUCCESS_CODE;
00992
00993
        if (cqe->res < 0)</pre>
00994
         {
            list_delete_node (&s->list, req);
LOG_END (" ");
00995
00996
00997
            return CRITICAL_CODE;
00998
00999
        if (req->client_socket == s->efds[0])
01000
01001
            io_uring_cge_seen (&s->rings[0], cge);
            list_delete_node (&s->list, req);
LOG_END (" ");
01002
01003
01004
            return ERROR_CODE;
01005
01006
        io_uring_cqe_seen (&s->rings[0], cqe);
01007
        switch (req->event_type)
01008
          {
          case EV_ACC:
01010
            handle_accept (s, cqe->res);
01011
             add_accept (s, client_addr, client_addr_len);
01012
             add_read (s, cqe->res);
             list_delete_node (&s->list, req);
01013
01014
            break;
          case EV_REA:
01015
01016
           handle_event_read (cqe, s, req);
01017
            break;
01018
          case EV_WRI:
01019
            handle_write (s, req->client_socket);
             list_delete_node (&s->list, req);
01020
01021
            break:
01022
01023
        LOG_END (" ");
01024
        return SUCCESS_CODE;
01025 }
01026
01027 // saurion_worker_master
01028 void
01029 saurion_worker_master (void *arg)
01030 {
        LOG_INIT (" ");
01031
        struct saurion *const s = (struct saurion *)arg;
01032
        struct sockaddr_in client_addr;
01033
01034
        socklen_t client_addr_len = sizeof (client_addr);
01035
01036
        add_efd (s, s->efds[0], 0);
01037
        add_accept (s, &client_addr, &client_addr_len);
01038
01039
        pthread mutex lock (&s->status m);
01040
        ++s->status;
01041
        pthread_cond_broadcast (&s->status_c);
01042
        pthread_mutex_unlock (&s->status_m);
01043
        while (1)
01044
         {
            int ret
01045
             = saurion_worker_master_loop_it (s, &client_addr, &client_addr_len);
if (ret == ERROR_CODE || ret == CRITICAL_CODE)
01046
01047
01048
              {
01049
                break;
              }
01050
01051
        pthread_mutex_lock (&s->status_m);
01052
         --s->status;
01054
        pthread_cond_signal (&s->status_c);
01055
        pthread_mutex_unlock (&s->status_m);
01056
        LOG_END (" ");
01057
        return;
01058 }
```

7.14 low_saurion.c 101

```
01060 // saurion_worker_slave_loop_it
01061 [[nodiscard]]
01062 static inline int
01063 saurion_worker_slave_loop_it (struct saurion *const s, const int sel)
01064 {
01065
01066
        struct io_uring ring = s->rings[sel];
01067
        struct io_uring_cqe *cqe = NULL;
01068
        add_efd (s, s->efds[sel], sel);
01069
01070
        int ret = io_uring_wait_cqe (&ring, &cqe);
        if (ret < 0)</pre>
01071
01072
            free (cqe);
LOG_END (" ");
01073
01074
            return CRITICAL_CODE;
01075
01076
01077
        struct request *req = (struct request *)cqe->user_data;
01078
        if (!req)
01079
            io_uring_cqe_seen (&ring, cqe);
LOG_END (" ");
01080
01081
01082
            return SUCCESS CODE;
01083
01084
        if (cqe->res < 0)</pre>
01085
          {
            list_delete_node (&s->list, req);
LOG_END (" ");
01086
01087
            return CRITICAL_CODE;
01088
01089
01090
        if (req->client_socket == s->efds[sel])
01091
01092
            io_uring_cqe_seen (&ring, cqe);
            list_delete_node (&s->list, req);
LOG_END (" ");
01093
01094
            return ERROR_CODE;
01095
01096
01097
        io_uring_cqe_seen (&ring, cqe);
01098
        switch (req->event_type)
01099
01100
          case EV REA:
01101
           handle_event_read (cqe, s, req);
01102
            break;
          case EV_WRI:
01103
01104
            handle_write (s, req->client_socket);
01105
            list_delete_node (&s->list, req);
01106
            break;
01107
        LOG_END (" ");
01108
        return SUCCESS_CODE;
01109
01110 }
01111
01112 // saurion_worker_slave
01113 void
01114 saurion_worker_slave (void *arg)
01115 {
01116
        LOG_INIT (" ");
01117
        struct saurion_wrapper *const ss = (struct saurion_wrapper *)arg;
        struct saurion *s = ss->s;
const int sel = ss->sel;
01118
01119
01120
        free (ss);
01121
01122
        add_efd (s, s->efds[sel], sel);
01123
01124
        pthread_mutex_lock (&s->status_m);
01125
        ++s->status;
        pthread_cond_broadcast (&s->status_c);
01126
01127
        pthread_mutex_unlock (&s->status_m);
01128
        while (1)
01129
         {
01130
            int res = saurion_worker_slave_loop_it (s, sel);
             if (res == ERROR_CODE || res == CRITICAL_CODE)
01131
01132
01133
                break;
01134
01135
01136
        pthread_mutex_lock (&s->status_m);
         --s->status;
01137
01138
        pthread cond signal (&s->status c);
        pthread_mutex_unlock (&s->status_m);
01139
        LOG_END (" ");
01140
01141
        return;
01142 }
01143
01144 // saurion_start
01145 [[nodiscard]]
```

```
01146 int
01147 saurion_start (struct saurion *const s)
01148 {
01149
        threadpool_init (s->pool);
        threadpool_add (s->pool, saurion_worker_master, s);
struct saurion_wrapper *ss = NULL;
for (uint32_t i = 1; i < s->n_threads; ++i)
01150
01151
01152
01153
01154
            ss = (struct saurion_wrapper *)malloc (sizeof (struct saurion_wrapper));
01155
            if (!ss)
             {
              return ERROR_CODE;
}
01156
01157
01158
            ss->s = s;
01159
01160
            ss->sel = i;
01161
            threadpool_add (s->pool, saurion_worker_slave, ss);
01162
01163
        pthread_mutex_lock (&s->status_m);
        while (s->status < (int)s->n_threads)
01164
        {
01165
01166
            pthread_cond_wait (&s->status_c, &s->status_m);
01167
01168
       pthread_mutex_unlock (&s->status_m);
01169
        return SUCCESS CODE;
01170 }
01171
01172 // saurion_stop
01173 void
01174 saurion_stop (const struct saurion *const s)
01175 {
01176
       uint64 t u = 1;
01177
        for (uint32_t i = 0; i < s->n_threads; ++i)
01178
01179
            while (write (s\rightarrow efds[i], \&u, sizeof(u)) < 0)
01180
                nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
01181
01182
01183
01184
       threadpool_wait_empty (s->pool);
01185 }
01186
01187 // saurion_destroy
01188 void
01189 saurion_destroy (struct saurion *const s)
01190 {
01191
        pthread_mutex_lock (&s->status_m);
01192
        while (s->status > 0)
01193
            pthread_cond_wait (&s->status_c, &s->status_m);
01194
01195
01196
        pthread_mutex_unlock (&s->status_m);
01197
        threadpool_destroy (s->pool);
01198
        for (uint32_t i = 0; i < s->n_threads; ++i)
01199
01200
            io_uring_queue_exit (&s->rings[i]);
01201
           pthread_mutex_destroy (&s->m_rings[i]);
01202
01203
        free (s->m_rings);
01204
        list_free (&s->list);
01205
        for (uint32_t i = 0; i < s->n_threads; ++i)
01206
        {
01207
            close (s->efds[i]);
01208
01209
        free (s->efds);
01210
        if (!s->ss)
01211
        {
01212
           close (s->ss);
          }
01213
01214
       free (s->rings);
01215
       pthread_mutex_destroy (&s->status_m);
01216
        pthread_cond_destroy (&s->status_c);
01217
        free (s);
01218 }
01219
01220 // saurion_send
01221 void
01222 saurion_send (struct saurion *const s, const int fd, const char *const msg)
01223 {
01224
        add_write (s, fd, msg, next (s));
01225 }
```

7.15 / w/saurion/saurion/src/main.c File Reference

```
#include <pthread.h>
#include <stdio.h>
Include dependency graph for main.c:
```

7.16 main.c

```
Go to the documentation of this file.
00001 #include <pthread.h> // for pthread_create, pthread_join, pthread_t 00002 #include <stdio.h> // for printf, fprintf, NULL, stderr
00003
00004 int counter = 0;
00005
00006 void *
00007 increment (void *arg)
00008 {
00009
        int id = *((int *)arg);
00010
        for (int i = 0; i < 100000; ++i)
00011
00012
00013
             if (i % 10000 == 0)
00014
                printf ("Thread %d at iteration %dn", id, i);
00015
00016
00018 printf ("Thread %d finished\n", id);
00019
        return NULL;
00020 }
00021
00022 int
00023 main ()
00024 {
00025 pthread_t t1;
00026
        pthread_t t2;
00027
        int id1 = 1;
int id2 = 2;
00028
00030
        printf ("Starting threads...\n");
00031
00032
        if (pthread_create (&t1, NULL, increment, &id1))
00033
00034
             fprintf (stderr, "Error creating thread 1\n");
00035
             return 1;
00036
00037
        if (pthread_create (&t2, NULL, increment, &id2))
00038
         {
00039
             fprintf (stderr, "Error creating thread 2\n");
00040
             return 1;
00041
00042
00043
        printf ("Waiting for thread 1 to join...\n");
00044
        if (pthread_join (t1, NULL))
00045
             fprintf (stderr, "Error joining thread 1\n"); return 2;
00046
00047
00048
00049
        printf ("Thread 1 joined\n");
00050
00051
        printf ("Waiting for thread 2 to join...\n");
00052
        if (pthread_join (t2, NULL))
00053
00054
             fprintf (stderr, "Error joining thread 2\n");
00055
00056
00057
        printf ("Thread 2 joined\n");
00058
        printf ("Final counter value: %d\n", counter);
00059
00060
        return 0;
```

7.17 /_w/saurion/saurion/src/saurion.cpp File Reference

```
#include "saurion.hpp"
#include "low_saurion.h"
```

Include dependency graph for saurion.cpp:

7.18 saurion.cpp

Go to the documentation of this file.

```
00001 #include "saurion.hpp"
00002
00003 #include "low_saurion.h" // for saurion, saurion_create, saurion_destroy
00004
00005 Saurion::Saurion (const uint32_t thds, const int sck) noexcept
00006 {
       this->s = saurion_create (thds);
00007
80000
       if (!this->s)
00010
            return;
00011
00012
       this->s->ss = sck;
00013 }
00014
00015 Saurion::~Saurion () { saurion_destroy (this->s); }
00016
00017 void
00018 Saurion::init () noexcept
00019 {
00020
        if (!saurion_start (this->s))
00021
         {
00022
            return;
00023
00024 }
00025
00026 void
00027 Saurion::stop () const noexcept
00029
        saurion_stop (this->s);
00030 }
00031
00032 Saurion *
00033 Saurion::on_connected (Saurion::ConnectedCb ncb, void *arg) noexcept
00034 {
00035 s->cb.on_connected = ncb;
00036 s->cb.on_connected_arg = arg;
00037 return this;
00038 }
00039
00040 Saurion *
00041 Saurion::on_readed (Saurion::ReadedCb ncb, void *arg) noexcept
00042 {
00043
       s->cb.on_readed = ncb;
00044 s->cb.on_readed_arg = arg;
00045 return this;
00046 }
00047
00048 Saurion *
00049 Saurion::on_wrote (Saurion::WroteCb ncb, void *arg) noexcept
00050 {
00051
       s->cb.on_wrote = ncb;
      s->cb.on_wrote_arg = arg;
00052
00053
       return this;
00054 }
00055
00056 Saurion *
00057 Saurion::on_closed (Saurion::ClosedCb ncb, void *arg) noexcept
00058 {
00059
       s->cb.on_closed = ncb;
00060
       s->cb.on_closed_arg = arg;
00061 return this;
00062 }
00063
00064 Saurion *
00065 Saurion::on_error (Saurion::ErrorCb ncb, void *arg) noexcept
00066 {
00067
       s->cb.on_error = ncb;
00068 s->cb.on_error_arg = arg;
00069
       return this;
00070 }
00071
00072 void
00073 Saurion::send (const int fd, const char *const msg) noexcept
00074 {
00075
        saurion_send (this->s, fd, msg);
00076 }
```

7.19 /_w/saurion/saurion/src/threadpool.c File Reference

```
#include "threadpool.h"
#include "config.h"
#include <nanologger.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
```

Include dependency graph for threadpool.c:

Classes

- struct task
- · struct threadpool

Macros

- #define TRUE 1
- #define FALSE 0

Functions

- struct threadpool * threadpool_create (size_t num_threads)
- struct threadpool * threadpool_create_default (void)
- void * threadpool_worker (void *arg)
- void threadpool_init (struct threadpool *pool)
- void threadpool add (struct threadpool *pool, void(*function)(void *), void *argument)
- void threadpool_stop (struct threadpool *pool)
- int threadpool_empty (struct threadpool *pool)
- void threadpool_wait_empty (struct threadpool *pool)
- void threadpool_destroy (struct threadpool *pool)

7.19.1 Macro Definition Documentation

7.19.1.1 FALSE

```
#define FALSE 0
```

Definition at line 9 of file threadpool.c.

7.19.1.2 TRUE

```
#define TRUE 1
```

Definition at line 8 of file threadpool.c.

7.19.2 Function Documentation

7.19.2.1 threadpool_worker()

```
void * threadpool_worker (
               void * arg )
Definition at line 107 of file threadpool.c.
00108 {
00110
        struct threadpool *pool = (struct threadpool *)arg;
00111
        while (TRUE)
00112
            pthread_mutex_lock (&pool->queue_lock);
while (pool->task_queue_head == NULL && !pool->stop)
00113
00114
00115
00116
                 pthread_cond_wait (&pool->queue_cond, &pool->queue_lock);
00117
00118
00119
             if (pool->stop && pool->task_queue_head == NULL)
00120
              {
                pthread_mutex_unlock (&pool->queue_lock);
00122
00123
00124
            struct task *task = pool->task_queue_head;
00125
            if (task != NULL)
00126
00127
00128
                 pool->task_queue_head = task->next;
00129
                 if (pool->task_queue_head == NULL)
00130
                  pool->task_queue_tail = NULL;
00131
00132
                 if (pool->task_queue_head == NULL)
00133
00134
                     pthread_cond_signal (&pool->empty_cond);
00135
00136
00137
            pthread_mutex_unlock (&pool->queue_lock);
00138
```

task->function (task->argument);

7.20 threadpool.c

LOG_END (" ");

return NULL;

pthread_exit (NULL);

00139

00140 00141

00142

00143

00145 00146

00147

00148 }

Go to the documentation of this file.

if (task != NULL)

free (task);

```
00001 #include "threadpool.h"
00002 #include "config.h"
                                     // for NUM CORES
00003 #include <nanologger.h> // for LOG_END, LOG_INIT
00004 #include <pthread.h> // for pthread_mutex_unlock, pthread_mutex_lock
00005 #include <stdio.h> // for perror
00006 #include <stdlib.h>
                                      // for free, malloc
00007
00008 #define TRUE 1
00009 #define FALSE 0
00010
00011 struct task
00012 {
00013
         void (*function) (void *);
        void *argument;
00014
00015
         struct task *next;
00016 };
00017
00018 struct threadpool
```

7.20 threadpool.c 107

```
00019 {
00020
        pthread_t *threads;
00021
        size_t num_threads;
00022
        struct task *task_queue_head;
        struct task *task_queue_tail;
00023
00024
        pthread_mutex_t queue_lock;
00025
        pthread_cond_t queue_cond;
00026
        pthread_cond_t empty_cond;
00027
        int stop;
00028
        int started;
00029 1:
00030
00031 struct threadpool *
00032 threadpool_create (size_t num_threads)
00033 {
00034
        LOG_INIT (" ");
        struct threadpool *pool = malloc (sizeof (struct threadpool));
00035
00036
        if (pool == NULL)
00037
            perror ("Failed to allocate threadpool"); LOG\_END (" ");
00038
00039
00040
             return NULL;
00041
00042
        if (num_threads < 3)</pre>
00043
          {
00044
            num_threads = 3;
00045
00046
        if (num_threads > NUM_CORES)
00047
00048
            num_threads = NUM_CORES;
00049
00050
00051
        pool->num_threads = num_threads;
00052
        pool->threads = malloc (sizeof (pthread_t) * num_threads);
00053
        if (pool->threads == NULL)
00054
00055
             perror ("Failed to allocate threads array");
             free (pool);
00056
00057
             LOG_END (" ");
00058
            return NULL;
00059
00060
00061
        pool->task_queue_head = NULL;
00062
        pool->task_queue_tail = NULL;
00063
        pool->stop = FALSE;
00064
        pool->started = FALSE;
00065
00066
        if (pthread_mutex_init (&pool->queue_lock, NULL) != 0)
00067
            perror ("Failed to initialize mutex");
00068
             free (pool->threads);
00069
            free (pool);
LOG_END (" ");
00070
00071
00072
             return NULL;
00073
00074
00075
        if (pthread_cond_init (&pool->queue_cond, NULL) != 0)
00076
         {
00077
            perror ("Failed to initialize condition variable");
00078
             pthread_mutex_destroy (&pool->queue_lock);
00079
             free (pool->threads);
            free (pool);
LOG_END (" ");
00080
00081
00082
            return NULL;
00083
00084
00085
        if (pthread_cond_init (&pool->empty_cond, NULL) != 0)
00086
          {
00087
            perror ("Failed to initialize empty condition variable");
            pthread_mutex_destroy (&pool->queue_lock);
pthread_cond_destroy (&pool->queue_cond);
00088
00089
00090
             free (pool->threads);
            free (pool);
LOG_END (" ");
00091
00092
00093
            return NULL;
00094
00095
00096
       LOG_END (" ");
00097
        return pool;
00098 }
00099
00100 struct threadpool *
00101 threadpool_create_default (void)
00102 {
00103
        return threadpool_create (NUM_CORES);
00104 }
00105
```

```
00106 void *
00107 threadpool_worker (void *arg)
00108 {
00109
       LOG INIT (" ");
        struct threadpool *pool = (struct threadpool *)arg;
00110
00111
       while (TRUE)
00112
00113
            pthread_mutex_lock (&pool->queue_lock);
00114
            while (pool->task_queue_head == NULL && !pool->stop)
00115
00116
               pthread_cond_wait (&pool->queue_cond, &pool->queue_lock);
00117
00118
00119
            if (pool->stop && pool->task_queue_head == NULL)
00120
               pthread_mutex_unlock (&pool->queue_lock);
00121
00122
                break:
             }
00123
00125
            struct task *task = pool->task_queue_head;
00126
            if (task != NULL)
00127
00128
               pool->task_queue_head = task->next;
00129
                if (pool->task gueue head == NULL)
00130
                 pool->task_queue_tail = NULL;
00131
00132
                if (pool->task_queue_head == NULL)
00133
00134
                   pthread_cond_signal (&pool->empty_cond);
                  }
00135
00136
00137
           pthread_mutex_unlock (&pool->queue_lock);
00138
00139
            if (task != NULL)
00140
               task->function (task->argument);
00141
00142
               free (task);
00144
00145 LOG_END (" ");
       pthread_exit (NULL);
00146
       return NULL;
00147
00148 }
00149
00150 void
00151 threadpool_init (struct threadpool *pool)
00152 {
       LOG_INIT (" ");
00153
        if (pool == NULL || pool->started)
00154
00155
            LOG_END (" ");
00156
00157
           return;
00158
00159
       for (size_t i = 0; i < pool->num_threads; i++)
00160
00161
           if (pthread_create (&pool->threads[i], NULL, threadpool_worker,
00162
                                (void *)pool)
00163
00164
               perror ("Failed to create thread");
00165
                pool->stop = TRUE;
00166
00167
               break;
00168
             }
00169
00170
       pool->started = TRUE;
00171
       LOG_END (" ");
00172 }
00173
00174 void
00175 threadpool_add (struct threadpool *pool, void (*function) (void *),
00176
                     void *argument)
00177 {
00178
       LOG_INIT (" ");
        if (pool == NULL || function == NULL)
00179
00180
        {
00181
          LOG_END (" ");
00182
           return;
00183
00184
        struct task *new_task = malloc (sizeof (struct task));
00185
00186
        if (new_task == NULL)
00187
         {
           perror ("Failed to allocate task");
LOG_END (" ");
00188
00189
00190
           return;
00191
00192
```

7.20 threadpool.c 109

```
00193
        new_task->function = function;
00194
        new_task->argument = argument;
00195
        new_task->next = NULL;
00196
00197
        pthread_mutex_lock (&pool->queue_lock);
00198
00199
        if (pool->task_queue_head == NULL)
00200
00201
            pool->task_queue_head = new_task;
00202
            pool->task_queue_tail = new_task;
00203
          }
00204
        else
00205
         {
00206
            pool->task_queue_tail->next = new_task;
00207
            pool->task_queue_tail = new_task;
00208
00209
        pthread_cond_signal (&pool->queue_cond);
00210
00211
        pthread_mutex_unlock (&pool->queue_lock);
00212
       LOG_END (" ");
00213 }
00214
00215 void
00216 threadpool_stop (struct threadpool \starpool)
00217 {
00218
        LOG_INIT (" ");
00219
        if (pool == NULL || !pool->started)
00220
            LOG_END (" ");
00221
00222
            return;
00223
00224
       threadpool_wait_empty (pool);
00225
00226
        pthread_mutex_lock (&pool->queue_lock);
00227
        pool->stop = TRUE;
        pthread_cond_broadcast (&pool->queue_cond);
00228
00229
        pthread_mutex_unlock (&pool->queue_lock);
00230
00231
        for (size_t i = 0; i < pool->num_threads; i++)
00232
00233
            pthread_join (pool->threads[i], NULL);
00234
        pool->started = FALSE;
00235
        LOG_END (" ");
00236
00237 }
00238
00239 int
00240 threadpool_empty (struct threadpool \starpool)
00241 {
00242
        LOG_INIT (" ");
        if (pool == NULL)
00243
00244
            LOG_END (" ");
00245
00246
            return TRUE;
00247
00248
       pthread mutex lock (&pool->queue lock);
00249
        int empty = (pool->task_queue_head == NULL);
00250
        pthread_mutex_unlock (&pool->queue_lock);
00251
        LOG_END (" ");
00252
        return empty;
00253 }
00254
00255 void
00256 threadpool_wait_empty (struct threadpool *pool)
00257 {
00258
        LOG_INIT (" ");
        if (pool == NULL)
00259
00260
         {
00261
            LOG_END (" ");
00262
            return;
00263
00264
        pthread_mutex_lock (&pool->queue_lock);
00265
        while (pool->task_queue_head != NULL)
00266
00267
            pthread_cond_wait (&pool->empty_cond, &pool->queue_lock);
00268
00269
        pthread_mutex_unlock (&pool->queue_lock);
00270
        LOG_END (" ");
00271 }
00272
00273 void
00274 threadpool_destroy (struct threadpool *pool)
00275 {
00276
        LOG_INIT (" ");
00277
        if (pool == NULL)
00278
00279
            LOG_END (" ");
```

```
return;
00280
00281
00282
            threadpool_stop (pool);
00283
00284
           pthread_mutex_lock (&pool->queue_lock);
struct task *task = pool->task_queue_head;
while (task != NULL)
00285
00286
00287
                 struct task *tmp = task;
task = task->next;
00288
00289
                 free (tmp);
00290
00291
00292
            pthread_mutex_unlock (&pool->queue_lock);
00293
           pthread_mutex_destroy (&pool->queue_lock);
pthread_cond_destroy (&pool->queue_cond);
pthread_cond_destroy (&pool->empty_cond);
00294
00295
00296
00297
00298
           free (pool->threads);
           free (pool);
LOG_END (" ");
00299
00300
00301 }
```

Index

/_w/saurion/saurion/include/linked_list.h, 55, 57 / w/saurion/saurion/include/low saurion.h, 57, 64	ClosedCb Saurion, 38
/_w/saurion/saurion/include/low_saurion_secret.h, 64,	ConnectedCb Saurion, 38
/w/saurion/saurion/include/saurion.hpp, 65, 66	cont rem
/_w/saurion/saurion/include/threadpool.h, 66, 67	chunk_params, 29
/_w/saurion/saurion/src/linked_list.c, 67, 70	cont sz
/_w/saurion/saurion/src/low_saurion.c, 72, 88	chunk_params, 29
/_w/saurion/saurion/src/main.c, 103	copy data
/_w/saurion/saurion/src/saurion.cpp, 103, 104	low saurion.c, 79
/_w/saurion/saurion/src/threadpool.c, 105, 106	create_node
POSIX_C_SOURCE	linked_list.c, 68
LowSaurion, 12	curr_iov
\sim Saurion	chunk_params, 30
Saurion, 39	curr_iov_off
	chunk_params, 30
add_accept	–
low_saurion.c, 75	dest
add_efd	chunk_params, 30
low_saurion.c, 76	dest_off
add_fd	chunk_params, 30
low_saurion.c, 76	dest_ptr
add_read	chunk_params, 30
low_saurion.c, 77	
add_read_continue	efds
low_saurion.c, 77	low_saurion.h, 58
add_write	saurion, 35
low_saurion.c, 78	empty_cond
allocate_iovec	threadpool, 52
LowSaurion, 12	ErrorCb
argument	Saurion, 39
task, 51	EV_ACC
	low_saurion.c, 74
calculate_max_iov_content	EV_ERR
low_saurion.c, 78	low_saurion.c, 74
children	EV_REA
Node, 31	low_saurion.c, 74
chunk_params, 29	EV_WAI
cont_rem, 29	low_saurion.c, 74
cont_sz, 29	EV_WRI
curr_iov, 30	low_saurion.c, 75
curr_iov_off, 30	event_type
dest, 30	request, 33
dest_off, 30	
dest_ptr, 30	FALSE
len, 30	threadpool.c, 105
max_iov_cont, 31	free_node
req, 31	linked_list.c, 68
client_socket	free_request
request, 33	LowSaurion, 13

function	linked_list.c, 70
task, 51	low saurion.c
	add accept, 75
handle_accept	add_efd, 76
low_saurion.c, 79	add fd, 76
handle_close	add_read, 77
low_saurion.c, 80	add_read_continue, 77
handle_error	add_write, 78
low_saurion.c, 80	calculate_max_iov_content, 78
handle_event_read	copy_data, 79
low_saurion.c, 80	EV ACC, 74
handle_new_message	EV_ERR, 74
low_saurion.c, 81	EV REA, 74
handle_partial_message	EV WAI, 74
low_saurion.c, 81	EV WRI, 75
handle_previous_message	handle_accept, 79
low_saurion.c, 82	handle_close, 80
handle_read	handle_error, 80
low_saurion.c, 82	handle_event_read, 80
handle_write	handle_new_message, 81
low_saurion.c, 83	handle_partial_message, 81
htonll	handle_previous_message, 82
low_saurion.c, 83	handle_read, 82
	handle_write, 83
init	htonll, 83
Saurion, 40	MAX, 75
initialize_iovec	MIN, 75
LowSaurion, 13	next, 84
iov	ntohll, 84
request, 33	prepare_destination, 84
iovec_count	read_chunk_free, 84
request, 33	saurion_worker_master, 85
	saurion_worker_master_loop_it, 85
len	saurion_worker_slave, 86
chunk_params, 30	saurion worker slave loop it, 87
linked_list.c	TIMEOUT_RETRY_SPEC, 88
create_node, 68	validate_and_update, 87
free_node, 68	low_saurion.h
list_delete_node, 69	efds, 58
list_free, 69	list, 59
list_insert, 69	m_rings, 59
list_mutex, 70	n_threads, 59
linked_list.h	next, 59
list_delete_node, 55	on_closed, 59
list_free, 56	on_closed_arg, 60
list_insert, 56	on_connected, 60
list	on_connected_arg, 60
low_saurion.h, 59	on_error, 60
saurion, 35	on_error_arg, 61
list_delete_node	on_readed, 61
linked_list.c, 69	on_readed_arg, 61
linked_list.h, 55	on_wrote, 62
list_free	on_wrote_arg, 62
linked_list.c, 69	pool, 62
linked_list.h, 56	rings, 62
list_insert	ss, 63
linked_list.c, 69	status, 63
linked_list.h, 56	status_c, 63
list_mutex	

status_m, 63	low_saurion.h, 60
LowSaurion, 9	Saurion, 40
_POSIX_C_SOURCE, 12	saurion::saurion_callbacks, 44
allocate_iovec, 12	saurion_callbacks, 47
free_request, 13	on_connected_arg
initialize_iovec, 13	low_saurion.h, 60
PACKING_SZ, 12	saurion::saurion_callbacks, 44
read_chunk, 15	saurion callbacks, 48
saurion_create, 16	on_error
saurion destroy, 18	low_saurion.h, 60
saurion_send, 19	Saurion, 41
saurion_set_socket, 19	saurion::saurion_callbacks, 44
saurion_start, 20	saurion callbacks, 48
saurion_start, 20	on error arg
set_request, 21	low_saurion.h, 61
Set_Tequest, 21	
m_rings	saurion::saurion_callbacks, 45
low_saurion.h, 59	saurion_callbacks, 48
	on_readed
saurion, 35 MAX	low_saurion.h, 61
	Saurion, 41
low_saurion.c, 75	saurion::saurion_callbacks, 45
max_iov_cont	saurion_callbacks, 48
chunk_params, 31	on_readed_arg
MIN	low_saurion.h, 61
low_saurion.c, 75	saurion::saurion_callbacks, 45
	saurion_callbacks, 49
n_threads	on wrote
low_saurion.h, 59	low_saurion.h, 62
saurion, 36	Saurion, 41
next	saurion::saurion_callbacks, 45
low_saurion.c, 84	saurion_callbacks, 49
low_saurion.h, 59	on_wrote_arg
Node, 32	low_saurion.h, 62
saurion, 36	saurion::saurion_callbacks, 46
task, 51	saurion_callbacks, 49
next iov	
request, 33	operator=
next_offset	Saurion, 41, 42
request, 33	DACKING 67
Node, 31	PACKING_SZ
children, 31	LowSaurion, 12
	pool
next, 32	low_saurion.h, 62
ptr, 32	saurion, <mark>36</mark>
size, 32	prepare_destination
ntohll	low_saurion.c, 84
low_saurion.c, 84	prev
num_threads	request, 34
threadpool, 52	prev_remain
	request, 34
on_closed	prev_size
low_saurion.h, 59	request, 34
Saurion, 40	ptr
saurion::saurion_callbacks, 43	Node, 32
saurion_callbacks, 47	. 1000, 02
on_closed_arg	queue_cond
low_saurion.h, 60	threadpool, 52
saurion::saurion_callbacks, 44	queue lock
saurion_callbacks, 47	threadpool, 52
on_connected	oaapooi, oz

read_chunk	on_connected_arg, 44
LowSaurion, 15	on_error, 44
read_chunk_free	on_error_arg, 45
low_saurion.c, 84	on_readed, 45
ReadedCb	on_readed_arg, 45
Saurion, 39	on_wrote, 45
req	on_wrote_arg, 46
chunk_params, 31	saurion_callbacks, 46
request, 32	on_closed, 47
client_socket, 33	on_closed_arg, 47
event_type, 33	on_connected, 47
iov, 33	on_connected_arg, 48
iovec_count, 33	on_error, 48
next_iov, 33	on_error_arg, 48
next_offset, 33	on_readed, 48
prev, 34	on_readed_arg, 49
prev_remain, 34	on_wrote, 49
prev_size, 34	on_wrote_arg, 49
rings	saurion_create
low_saurion.h, 62	LowSaurion, 16
saurion, 36	saurion_destroy
S	LowSaurion, 18
Saurion, 42	saurion_send
saurion_wrapper, 50	LowSaurion, 19
Saurion, 37	saurion_set_socket
~Saurion, 39	LowSaurion, 19
ClosedCb, 38	saurion_start
ConnectedCb, 38	LowSaurion, 20
ErrorCb, 39	saurion_stop
init, 40	LowSaurion, 21
on closed, 40	saurion_worker_master
on connected, 40	low_saurion.c, 85
on error, 41	saurion_worker_master_loop_it
on_readed, 41	low_saurion.c, 85
on wrote, 41	saurion_worker_slave
operator=, 41, 42	low_saurion.c, 86
ReadedCb, 39	saurion_worker_slave_loop_it
s, 42	low_saurion.c, 87
Saurion, 39, 40	saurion_wrapper, 50
send, 42	s, 50
stop, 42	sel, 50
WroteCb, 39	sel
saurion, 34	saurion_wrapper, 50
efds, 35	send
list, 35	Saurion, 42
m_rings, 35	set_request
n_threads, 36	LowSaurion, 21
next, 36	size
pool, 36	Node, 32
rings, 36	SS
ss, 36	low_saurion.h, 63
	saurion, 36
status, 37 status_c, 37	started
status_m, 37	threadpool, 52
saurion::saurion_callbacks, 43	status
on_closed, 43	low_saurion.h, 63
on_closed_arg, 44	saurion, 37
-	status_c
on_connected, 44	

low_saurion.h, 63 saurion, 37	threadpool, 53 TIMEOUT_RETRY_SPEC
status_m	low_saurion.c, 88
low_saurion.h, 63	TRUE
saurion, 37	threadpool.c, 105
stop	
Saurion, 42	validate_and_update
threadpool, 52	low_saurion.c, 87
	W . O
task, 50	WroteCb
argument, 51	Saurion, 39
function, 51	
next, 51	
task_queue_head	
threadpool, 53	
task_queue_tail	
threadpool, 53	
ThreadPool, 23	
threadpool_add, 23	
threadpool_create, 24	
threadpool_create_default, 25	
threadpool_destroy, 25	
threadpool_empty, 25	
threadpool_init, 26	
threadpool_stop, 26	
threadpool_wait_empty, 27	
threadpool, 51	
empty_cond, 52	
num_threads, 52	
queue_cond, 52	
queue_lock, 52	
• —	
started, 52	
stop, 52	
task_queue_head, 53	
task_queue_tail, 53	
threads, 53	
threadpool.c	
FALSE, 105	
threadpool_worker, 106	
TRUE, 105	
threadpool_add	
ThreadPool, 23	
threadpool_create	
ThreadPool, 24	
threadpool_create_default	
ThreadPool, 25	
threadpool_destroy	
ThreadPool, 25	
threadpool_empty	
ThreadPool, 25	
threadpool_init	
ThreadPool, 26	
threadpool_stop	
ThreadPool, 26	
threadpool_wait_empty	
ThreadPool, 27	
threadpool_worker	
threadpool.c, 106	
threads	
=	