Saurion

Generated by Doxygen 1.9.4

1 Todo List	1
2 Module Index	3
2.1 Modules	 3
3 Class Index	5
3.1 Class List	 5
4 File Index	7
4 File Index 4.1 File List	7
4.11 lie List	 ,
5 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()	 18
5.1.3.6 saurion_destroy()	 20
5.1.3.7 saurion_send()	 20
5.1.3.8 saurion_set_socket()	 22
5.1.3.9 saurion_start()	 23
5.1.3.10 saurion_stop()	 23
5.1.3.11 set_request()	 24
5.2 ThreadPool	25
5.2.1 Detailed Description	 26
5.2.2 Function Documentation	 26
5.2.2.1 threadpool_add()	 26
5.2.2.2 threadpool_create()	 26
5.2.2.3 threadpool_create_default()	 27
5.2.2.4 threadpool_destroy()	28
5.2.2.5 threadpool_empty()	 28
5.2.2.6 threadpool_init()	28
5.2.2.7 threadpool_stop()	29
5.2.2.8 threadpool_wait_empty()	29
6 Class Documentation	31
6.1 Node Struct Reference	3 1
6.1.1 Detailed Description	31
O Dotaliou Douoliption	 J 1

6.1.2 Member Data Documentation	31
6.1.2.1 children	31
6.1.2.2 next	31
6.1.2.3 ptr	32
6.1.2.4 size	32
6.2 request Struct Reference	32
6.2.1 Detailed Description	32
6.2.2 Member Data Documentation	32
6.2.2.1 client_socket	32
6.2.2.2 event_type	33
6.2.2.3 iov	33
6.2.2.4 iovec_count	33
6.2.2.5 next_iov	33
6.2.2.6 next_offset	33
6.2.2.7 prev	33
6.2.2.8 prev_remain	34
6.2.2.9 prev_size	34
6.3 saurion Struct Reference	34
6.3.1 Detailed Description	35
6.3.2 Member Data Documentation	35
6.3.2.1 efds	35
6.3.2.2 list	35
6.3.2.3 m_rings	35
6.3.2.4 n_threads	35
6.3.2.5 next	36
6.3.2.6 pool	36
6.3.2.7 rings	36
6.3.2.8 ss	36
6.3.2.9 status	36
6.3.2.10 status_c	37
6.3.2.11 status_m	37
6.4 Saurion Class Reference	37
6.4.1 Detailed Description	38
6.4.2 Member Typedef Documentation	38
6.4.2.1 ClosedCb	38
6.4.2.2 ConnectedCb	38
6.4.2.3 ErrorCb	38
6.4.2.4 ReadedCb	38
6.4.2.5 WroteCb	39
6.4.3 Constructor & Destructor Documentation	39
6.4.3.1 Saurion() [1/3]	39
6.4.3.2 ~Saurion()	39

6.4.3.3 Saurion() [2/3]	39
6.4.3.4 Saurion() [3/3]	39
6.4.4 Member Function Documentation	40
6.4.4.1 init()	40
6.4.4.2 on_closed()	40
6.4.4.3 on_connected()	40
6.4.4.4 on_error()	40
6.4.4.5 on_readed()	41
6.4.4.6 on_wrote()	41
6.4.4.7 operator=() [1/2]	41
6.4.4.8 operator=() [2/2]	41
6.4.4.9 send()	41
6.4.4.10 stop()	42
6.4.5 Member Data Documentation	42
6.4.5.1 s	42
6.5 saurion::saurion_callbacks Struct Reference	42
6.5.1 Detailed Description	43
6.5.2 Member Data Documentation	43
6.5.2.1 on_closed	43
6.5.2.2 on_closed_arg	43
6.5.2.3 on_connected	43
6.5.2.4 on_connected_arg	44
6.5.2.5 on_error	44
6.5.2.6 on_error_arg	44
6.5.2.7 on_readed	44
6.5.2.8 on_readed_arg	45
6.5.2.9 on_wrote	45
6.5.2.10 on_wrote_arg	45
6.6 saurion_callbacks Struct Reference	46
6.6.1 Detailed Description	46
6.6.2 Member Data Documentation	46
6.6.2.1 on_closed	46
6.6.2.2 on_closed_arg	47
6.6.2.3 on_connected	47
6.6.2.4 on_connected_arg	47
6.6.2.5 on_error	47
6.6.2.6 on_error_arg	48
6.6.2.7 on_readed	48
6.6.2.8 on_readed_arg	48
6.6.2.9 on_wrote	48
6.6.2.10 on_wrote_arg	49
6.7 saurion, wrapper Struct Reference	49

	6.7.1 Detailed Description	49
	6.7.2 Member Data Documentation	49
	6.7.2.1 s	49
	6.7.2.2 sel	50
	6.8 task Struct Reference	50
	6.8.1 Detailed Description	50
	6.8.2 Member Data Documentation	50
	6.8.2.1 argument	50
	6.8.2.2 function	50
	6.8.2.3 next	51
	6.9 threadpool Struct Reference	51
	6.9.1 Detailed Description	51
	6.9.2 Member Data Documentation	51
	6.9.2.1 empty_cond	51
	6.9.2.2 num_threads	52
	6.9.2.3 queue_cond	52
	6.9.2.4 queue_lock	52
	6.9.2.5 started	52
	6.9.2.6 stop	52
	6.9.2.7 task_queue_head	52
	6.9.2.8 task_queue_tail	52
	6.9.2.9 threads	52
7	File Documentation	53
,		53
		53
		53
		54
	v	54
		55
	7.3 /w/saurion/saurion/include/low_saurion.h File Reference	55
	7.3.1 Variable Documentation	00
	7.0.1 Variable Becamentation	56
	7 3 1 1 efds	56 57
	7.3.1.1 efds	57
	7.3.1.2 list	57 57
	7.3.1.2 list	57 57 57
	7.3.1.2 list	57 57 57 57
	7.3.1.2 list	57 57 57 57
	7.3.1.2 list	57 57 57 57 57
	7.3.1.2 list	57 57 57 57 57 57 58
	7.3.1.2 list	57 57 57 57 57 57 58 58
	7.3.1.2 list	57 57 57 57 57 57 58

7.3.1.11 on_error_arg
7.3.1.12 on_readed
7.3.1.13 on_readed_arg
7.3.1.14 on_wrote
7.3.1.15 on_wrote_arg
7.3.1.16 pool
7.3.1.17 rings
7.3.1.18 ss
7.3.1.19 status
7.3.1.20 status_c
7.3.1.21 status_m
7.4 low_saurion.h
7.5 /w/saurion/saurion/include/low_saurion_secret.h File Reference
7.6 low_saurion_secret.h
7.7 /w/saurion/saurion/include/saurion.hpp File Reference
7.8 saurion.hpp
7.9 /w/saurion/saurion/include/threadpool.h File Reference
7.10 threadpool.h
7.11 /w/saurion/src/linked_list.c File Reference
7.11.1 Function Documentation
7.11.1.1 create_node()
7.11.1.2 free_node()
7.11.1.3 list_delete_node()
7.11.1.4 list_free()
7.11.1.5 list_insert()
7.11.2 Variable Documentation
7.11.2.1 list_mutex
7.12 linked_list.c
7.13 /w/saurion/src/low_saurion.c File Reference
7.13.1 Macro Definition Documentation
7.13.1.1 EV_ACC
7.13.1.2 EV_ERR
7.13.1.3 EV_REA
7.13.1.4 EV_WAI
7.13.1.5 EV_WRI
7.13.1.6 MAX
7.13.1.7 MIN
7.13.2 Function Documentation
7.13.2.1 add_accept()
7.13.2.2 add_efd()
7.13.2.3 add_fd()
7.13.2.4 add_read()

7.13.2.5 add_read_continue()	75
7.13.2.6 add_write()	75
7.13.2.7 handle_accept()	76
7.13.2.8 handle_close()	76
7.13.2.9 handle_error()	76
7.13.2.10 handle_read()	77
7.13.2.11 handle_write()	77
7.13.2.12 htonll()	77
7.13.2.13 next()	78
7.13.2.14 ntohll()	78
7.13.2.15 saurion_worker_master()	78
7.13.2.16 saurion_worker_master_loop_it()	79
7.13.2.17 saurion_worker_slave()	80
7.13.2.18 saurion_worker_slave_loop_it()	80
7.13.3 Variable Documentation	81
7.13.3.1 print_mutex	81
7.13.3.2 TIMEOUT_RETRY_SPEC	81
7.14 low_saurion.c	82
7.15 /w/saurion/saurion/src/main.c File Reference	94
7.16 main.c	94
7.17 /w/saurion/saurion/src/saurion.cpp File Reference	95
7.18 saurion.cpp	95
7.19 /w/saurion/saurion/src/threadpool.c File Reference	96
7.19.1 Macro Definition Documentation	97
7.19.1.1 FALSE	97
7.19.1.2 TRUE	97
7.19.2 Function Documentation	97
7.19.2.1 threadpool worker()	98
7.20 threadpool.c	98
Index	103

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	 							 										 			25

4 Module Index

Chapter 3

Class Index

3.1 Class List

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

Node .		31
request		32
saurion		
	Main structure for managing io_uring and socket events	34
Saurion		37
saurion::	saurion_callbacks	
	Structure containing callback functions to handle socket events	42
saurion_	callbacks	
	Structure containing callback functions to handle socket events	46
saurion_	wrapper	49
task		50
threadpo	ol	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	53
/w/saurion/saurion/include/low_saurion.h	55
/w/saurion/saurion/include/low_saurion_secret.h	62
/w/saurion/saurion/include/saurion.hpp	63
/_w/saurion/saurion/include/threadpool.h	ô4
/w/saurion/src/linked_list.c	ô5
/w/saurion/src/low_saurion.c	70
/_w/saurion/src/main.c	94
/_w/saurion/src/saurion.cpp	95
/ w/saurion/saurion/src/threadpool.c	96

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	- 1	Footer
	(64 bits:	9000)		(Message Data)	- 1	(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 159 of file low saurion.c.

```
00161 {
00162
        if (!iov || !chd_ptr)
00163
         {
00164
            return ERROR_CODE;
00165
00166
        iov->iov_base = malloc (CHUNK_SZ);
00167
        if (!iov->iov_base)
00168
         {
00169
            return ERROR_CODE;
00170
        iov->iov_len = (pos == (amount - 1) ? (size % CHUNK_SZ) : CHUNK_SZ);
if (iov->iov_len == 0)
00171
00172
00173
        {
00174
            iov->iov_len = CHUNK_SZ;
00175
00176
        chd_ptr[pos] = iov->iov_base;
00177 return SUCCESS_CODE;
00178 }
```

5.1.3.2 free_request()

Definition at line 91 of file low_saurion.c.

```
00092 {
         if (children_ptr)
00094
         {
00095
             free (children_ptr);
00096
             children_ptr = NULL;
00097
00098
        for (size_t i = 0; i < amount; ++i)</pre>
00099
             free (req->iov[i].iov_base);
00100
             req->iov[i].iov_base = NULL;
00101
00102
00103
        free (req);
00104
        req = NULL;
00104 feq - NOLL,

00105 free (children_ptr);

00106 children_ptr = NULL;
00107 }
```

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 iovec 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 iovec, 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 111 of file low_saurion.c.

```
00113 {
        if (!iov || !iov->iov_base)
00114
00115
         {
00116
             return ERROR_CODE;
00117
00118
           (msg)
00119
             size_t len = iov->iov_len;
00120
             char *dest = (char *)iov->iov_base;
char *orig = (char *)msg + pos * CHUNK_SZ;
00121
00122
00123
             size_t cpy_sz = 0;
00124
             if (h)
00125
                 if (pos == 0)
00126
00127
00128
                      uint64_t send_size = htonll (size);
00129
                      memcpy (dest, &send_size, sizeof (uint64_t));
```

```
dest += sizeof (uint64_t);
00131
                    len -= sizeof (uint64_t);
00132
00133
                else
00134
                  {
                    orig -= sizeof (uint64_t);
00135
00136
00137
                if ((pos + 1) == amount)
00138
                     --len;
00139
                    cpy_sz = (len < size ? len : size);
00140
00141
                    dest[cpy_sz] = 0;
00142
00143
00144
            cpy_sz = (len < size ? len : size);
00145
            memcpy (dest, orig, cpy_sz);
            dest += cpy_sz;
size_t rem = CHUNK_SZ - (dest - (char *)iov->iov_base);
00146
00147
00148
            memset (dest, 0, rem);
00149
00150
00151
00152
            memset ((char *)iov->iov_base, 0, CHUNK_SZ);
00153
00154
        return SUCCESS_CODE;
00155 }
```

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 428 of file low saurion.c.

```
00429 {
        if (req->iovec_count == 0)
00431
         {
00432
            return ERROR_CODE;
00433
00434
00435
        size_t max_iov_cont = 0; //< Total size of request</pre>
00436
        for (size_t i = 0; i < req->iovec_count; ++i)
00437
00438
            max_iov_cont += req->iov[i].iov_len;
00439
00440
        size t cont sz = 0:
00441
        size t cont rem = 0;
        size_t curr_iov = 0;
00442
00443
        size_t curr_iov_off = 0;
        size_t dest_off = 0;
void *dest_ptr = NULL;
00444
00445
00446
        if (req->prev && req->prev_size && req->prev_remain)
00447
        {
00448
           cont_sz = req->prev_size;
00449
            cont_rem = req->prev_remain;
            curr_iov = 0;
curr_iov_off = 0;
00450
00451
00452
            dest_off = cont_sz - cont_rem;
00453
            if (cont_rem <= max_iov_cont)</pre>
00454
             {
00455
                *dest = req->prev;
                dest_ptr = *dest;
req->prev = NULL;
00456
00457
00458
                req->prev_size = 0;
                req->prev_remain = 0;
00459
00460
00461
            else
00462
00463
                dest_ptr = req->prev;
00464
                *dest = NULL;
00465
00466
00467
        else if (req->next_iov || req->next_offset)
00468
00469
            curr_iov = req->next_iov;
00470
            curr_iov_off = req->next_offset;
00471
            cont_sz = *(
00472
                (size_t *)(((uint8_t *)req->iov[curr_iov].iov_base) + curr_iov_off));
            cont_sz = ntohll (cont_sz);
00474
            curr_iov_off += sizeof (uint64_t);
            cont_rem = cont_sz;
dest_off = cont_sz - cont_rem;
00475
00476
00477
            if ((curr_iov_off + cont_rem + 1) <= max_iov_cont)</pre>
00478
             {
00479
                 *dest = malloc (cont_sz);
00480
                 dest_ptr = *dest;
```

```
00481
00482
00483
                req->prev = malloc (cont_sz);
dest_ptr = req->prev;
*dest = NULL;
*len = 0;
00484
00485
00486
00487
00488
00489
00490
       else
        {
00491
           curr_iov = 0;
00492
            curr_iov_off = 0;
00493
00494
            cont_sz = *(
00495
                (size_t *)(((uint8_t *)req->iov[curr_iov].iov_base) + curr_iov_off));
00496
            cont_sz = ntohll (cont_sz);
00497
            curr_iov_off += sizeof (uint64_t);
00498
            cont_rem = cont_sz;
            dest_off = cont_sz - cont_rem;
00499
00500
            if (cont_rem <= max_iov_cont)</pre>
00501
00502
                *dest = malloc (cont_sz);
00503
                dest_ptr = *dest;
00504
00505
            else
00506
             {
00507
                req->prev = malloc (cont_sz);
00508
                dest_ptr = req->prev;
00509
                *dest = NULL;
00510
00511
00512
       size t curr iov msg rem = 0;
00513
00514
        uint8_t ok = 1UL;
00515
        while (1)
00516
00517
            curr iov msg rem
                = MIN (cont_rem, (req->iov[curr_iov].iov_len - curr_iov_off));
00519
            memcpy ((uint8_t *)dest_ptr + dest_off,
00520
                    ((uint8_t *)req->iov[curr_iov].iov_base) + curr_iov_off,
00521
                    curr_iov_msg_rem);
            dest_off += curr_iov_msg_rem;
00522
            curr_iov_off += curr_iov_msg_rem;
00523
00524
            cont_rem -= curr_iov_msg_rem;
            if (cont\_rem <= \overline{0})
00525
00526
00527
                if (*(((uint8_t *)req->iov[curr_iov].iov_base) + curr_iov_off) != 0)
00528
                    ok = OUL;
00529
00530
                *len = cont_sz;
00531
00532
                ++curr_iov_off;
00533
                break;
00534
00535
            if (curr_iov_off >= (req->iov[curr_iov].iov_len))
00536
              {
                ++curr_iov;
00538
                if (curr_iov == req->iovec_count)
00539
00540
                    break;
                  }
00541
00542
                curr_iov_off = 0;
00543
              }
00544
         }
00545
00546
        if (req->prev)
00547
00548
            req->prev_size = cont_sz;
00549
            req->prev_remain = cont_rem;
            *dest = NULL;
00551
            len = 0;
00552
00553
        else
00554
         {
00555
            req->prev size = 0;
00556
            req->prev_remain = 0;
00557
00558
        if (curr_iov < req->iovec_count)
00559
            uint64_t next_sz = *(uint64_t *)(((uint8_t *)req->iov[curr_iov].iov_base)
00560
00561
                                               + curr iov off);
00562
            if ((req->iov[curr_iov].iov_len > curr_iov_off) && next_sz)
00563
00564
                req->next_iov = curr_iov;
00565
                req->next_offset = curr_iov_off;
00566
00567
            else
```

```
{
00569
                req->next_iov = 0;
00570
                req->next_offset = 0;
00571
00572
          }
00573
00574
        if (ok)
00575
        {
00576
            return SUCCESS_CODE;
00577
00578
        free (dest_ptr);
00579
        dest_ptr = NULL;
        *dest = NULL;
*len = 0;
00580
00581
00582
        req->next_iov = 0;
00583
        req->next_offset = 0;
        for (size_t i = curr_iov; i < req->iovec_count; ++i)
00584
00585
00586
            for (size_t j = curr_iov_off; j < req->iov[i].iov_len; ++j)
00587
00588
                uint8_t foot = *((uint8_t *)req->iov[i].iov_base) + j;
00589
                if (foot == 0)
00590
                 {
                    req->next_iov = i;
req->next_offset = (j + 1) % req->iov[i].iov_len;
00591
00592
00593
                    return ERROR_CODE;
00594
00595
00596
          }
00597
       return ERROR CODE:
00598 }
```

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 707 of file low_saurion.c.

```
00708 {
00709
        LOG_INIT (" ");
00710
        struct saurion *p = (struct saurion *)malloc (sizeof (struct saurion));
00711
        if (!p)
       {
00712
00713
            LOG_END (" ");
00714
            return NULL;
00715
00716
       int ret = 0;
        ret = pthread_mutex_init (&p->status_m, NULL);
00718
       if (ret)
00719
        {
           free (p);
LOG_END (" ");
00720
00721
00722
            return NULL;
00723
00724
       ret = pthread_cond_init (&p->status_c, NULL);
```

```
00725
        if (ret)
00726
        {
            free (p);
LOG_END (" ");
00727
00728
00729
            return NULL;
00730
00731
        p->m_rings
00732
            = (pthread_mutex_t *) malloc (n_threads * sizeof (pthread_mutex_t));
        if (!p->m_rings)
00733
00734
00735
            free (p);
            LOG_END (" ");
00736
00737
            return NULL;
00738
00739
        for (uint32_t i = 0; i < n_threads; ++i)</pre>
00740
            pthread_mutex_init (&(p->m_rings[i]), NULL);
00741
00742
        p->ss = 0;
00744
        n_threads = (n_threads < 2 ? 2 : n_threads);</pre>
00745
        n_threads = (n_threads > NUM_CORES ? NUM_CORES : n_threads);
00746
        p->n_threads = n_threads;
00747
        p->status = 0;
        p->list = NULL;
00748
00749
        p->cb.on_connected = NULL;
00750
        p->cb.on_connected_arg = NULL;
00751
        p->cb.on_readed = NULL;
00752
        p->cb.on_readed_arg = NULL;
00753
        p->cb.on_wrote = NULL;
        p->cb.on_wrote_arg = NULL;
00754
        p->cb.on_closed = NULL;
00755
00756
        p->cb.on_closed_arg = NULL;
00757
        p->cb.on_error = NULL;
00758
        p->cb.on_error_arg = NULL;
        p->next = 0;
p->efds = (int *)malloc (sizeof (int) * p->n_threads);
00759
00760
00761
        if (!p->efds)
00762
00763
            free (p->m_rings);
00764
            free (p);
LOG_END (" ");
00765
            return NULL;
00766
00767
00768
        for (uint32_t i = 0; i < p->n_threads; ++i)
00769
            p->efds[i] = eventfd (0, EFD_NONBLOCK);
00770
00771
            if (p->efds[i] == ERROR_CODE)
00772
              {
00773
                for (uint32_t j = 0; j < i; ++j)
00774
00775
                    close (p->efds[j]);
00776
00777
                free (p->efds);
00778
                free (p->m_rings);
00779
                free (p);
LOG_END (" ");
00780
00781
                return NULL;
00782
00783
        p->rings
00784
00785
            = (struct io_uring *)malloc (sizeof (struct io_uring) * p->n_threads);
00786
        if (!p->rings)
00787
          {
00788
            for (uint32_t j = 0; j < p->n_threads; ++j)
00789
00790
                close (p->efds[j]);
00791
              }
00792
            free (p->efds):
            free (p->m_rings);
00793
            free (p);
LOG_END (" ");
00794
00795
00796
            return NULL;
00797
00798
        for (uint32_t i = 0; i < p->n_threads; ++i)
00799
00800
            memset (&p->rings[i], 0, sizeof (struct io_uring));
00801
            ret = io_uring_queue_init (SAURION_RING_SIZE, &p->rings[i], 0);
00802
            if (ret)
00803
              {
00804
                for (uint32_t j = 0; j < p->n_threads; ++j)
00805
                  {
00806
                    close (p->efds[j]);
00807
00808
                free (p->efds);
00809
                free (p->rings);
                free (p->m_rings);
00810
00811
                free (p);
```

5.1.3.6 saurion_destroy()

```
void saurion_destroy ( struct \ saurion * s )
```

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 1059 of file low_saurion.c.

```
01060 {
01061
        pthread mutex lock (&s->status m);
01062
        while (s->status > 0)
01063
         {
01064
             pthread_cond_wait (&s->status_c, &s->status_m);
01065
01066
        pthread_mutex_unlock (&s->status_m);
01067
        threadpool_destroy (s->pool);
        for (uint32_t i = 0; i < s->n_threads; ++i)
01068
01069
01070
             io_uring_queue_exit (&s->rings[i]);
01071
            pthread_mutex_destroy (&s->m_rings[i]);
01072
        free (s->m_rings);
list_free (&s->list);
01073
01074
01075
        for (uint32_t i = 0; i < s->n_threads; ++i)
01076
01077
             close (s->efds[i]);
01078
01079
        free (s->efds);
        if (!s->ss)
01080
01081
01082
             close (s->ss);
01083
01084
        free (s->rings);
        pthread_mutex_destroy (&s->status_m);
pthread_cond_destroy (&s->status_c);
01085
01086
01087
        free (s);
01088 }
```

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 1091 of file low saurion.c.

```
01092 {
01093     add_write (s, fd, msg, next (s));
01094 }
```

5.1.3.8 saurion_set_socket()

```
\begin{array}{c} \text{int saurion\_set\_socket (} \\ \text{int } p \text{ )} \end{array}
```

Creates a socket.

Creates and sets a socket, ready for saurion configuration.

Parameters



Returns

result of socket creation.

Definition at line 670 of file low_saurion.c.

```
00671 {
00672
         int sock = 0;
        struct sockaddr_in srv_addr;
00673
00674
00675
         sock = socket (PF_INET, SOCK_STREAM, 0);
00676
         if (sock < 1)</pre>
00677
00678
             return ERROR_CODE;
          }
00679
00680
00681
         int enable = 1;
00682
         if (setsockopt (sock, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof (int)) < 0)</pre>
00683
         {
00684
             return ERROR_CODE;
00685
00686
        memset (&srv_addr, 0, sizeof (srv_addr));
srv_addr.sin_family = AF_INET;
srv_addr.sin_port = htons (p);
00687
00689
00690
         srv_addr.sin_addr.s_addr = htonl (INADDR_ANY);
00691
00692
         if (bind (sock, (const struct sockaddr \star)&srv_addr, sizeof (srv_addr)) < 0)
00693
         {
00694
             return ERROR_CODE;
00695
00696
00697
         if (listen (sock, ACCEPT_QUEUE) < 0)</pre>
00698
00699
             return ERROR_CODE;
00701
00702
        return sock;
00703 }
```

5.1.3.9 saurion_start()

```
int saurion_start ( struct \ saurion \ * \ s \ )
```

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 1018 of file low_saurion.c.

```
01019 +
        pthread_mutex_init (&print_mutex, NULL);
threadpool_init (s->pool);
threadpool_add (s->pool, saurion_worker_master, s);
01020
01021
01022
01023
        struct saurion_wrapper *ss = NULL;
01024
        for (uint32_t i = 1; i < s->n_threads; ++i)
01025
01026
             ss = (struct saurion_wrapper *)malloc (sizeof (struct saurion_wrapper));
01027
             if (!ss)
             {
01029
                 return ERROR_CODE;
              }
01030
01031
            ss->s = s;
ss->sel = i;
01032
01033
             threadpool_add (s->pool, saurion_worker_slave, ss);
01034
01035
       pthread_mutex_lock (&s->status_m);
01036
        while (s->status < (int)s->n_threads)
01037
01038
             pthread_cond_wait (&s->status_c, &s->status_m);
01039
01040
       pthread_mutex_unlock (&s->status_m);
01041
        return SUCCESS_CODE;
01042 }
```

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 1045 of file low_saurion.c.

```
01046 {
01047
       uint64_t u = 1;
        for (uint32_t i = 0; i < s->n_threads; ++i)
01048
01049
           while (write (s->efds[i], \&u, sizeof(u)) < 0)
01050
01051
01052
               nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
01053
01054
01055
       threadpool_wait_empty (s->pool);
01056 }
```

5.1.3.11 set_request()

```
int set_request (
          struct request ** r,
           struct Node ** 1,
           size_t s,
           const void * m,
           uint8_t h ) [private]
```

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 182 of file low_saurion.c.

5.2 ThreadPool 25

```
00184 {
00185
        uint64_t full_size = s;
00186
        if (h)
00187
            full_size += (sizeof (uint64_t) + sizeof (uint8_t));
00188
00189
00190
        size_t amount = full_size / CHUNK_SZ;
00191
        amount = amount + (full_size % CHUNK_SZ == 0 ? 0 : 1);
00192
        struct request *temp = (struct request *)malloc (
00193
            sizeof (struct request) + sizeof (struct iovec) * amount);
        if (!temp)
00194
00195
         {
00196
            return ERROR_CODE;
00197
00198
        <u>if</u> (!*r)
00199
00200
           *r = temp;
            (*r)->prev = NULL;
00201
00202
            (*r)->prev_size = 0;
00203
            (*r)->prev_remain = 0;
00204
            (*r) \rightarrow next_iov = 0;
00205
            (*r)->next_offset = 0;
00206
00207
       else
00208
         {
            temp->client_socket = (*r)->client_socket;
00210
            temp->event_type = (*r)->event_type;
00211
            temp->prev = (*r)->prev;
00212
            temp->prev_size = (*r)->prev_size;
00213
            temp->prev_remain = (*r)->prev_remain;
00214
            temp->next iov = (*r)->next iov;
00215
            temp->next_offset = (*r)->next_offset;
00216
            *r = temp;
00217
00218
       struct request *req = *r;
00219
        req->iovec_count = (int)amount;
00220
        void **children_ptr = (void **)malloc (amount * sizeof (void *));
        if (!children_ptr)
00222
         {
00223
            free_request (req, children_ptr, 0);
00224
            return ERROR_CODE;
00225
        for (size_t i = 0; i < amount; ++i)</pre>
00226
00227
00228
            if (!allocate_iovec (&req->iov[i], amount, i, full_size, children_ptr))
00229
00230
                free_request (req, children_ptr, amount);
00231
                return ERROR_CODE;
00232
00233
            if (!initialize iovec (&reg->iov[i], amount, i, m, s, h))
00234
              {
00235
                free_request (req, children_ptr, amount);
00236
                return ERROR_CODE;
00237
00238
00239
        if (list insert (l, reg, amount, children ptr))
00240
00241
            free_request (req, children_ptr, amount);
00242
            return ERROR_CODE;
00243
00244
       free (children ptr);
00245
        return SUCCESS_CODE;
00246 }
```

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()

```
void threadpool\_add (
              struct threadpool * pool,
              void(*)(void *) function,
              void * argument )
Definition at line 175 of file threadpool.c.
00177 {
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
           perror ("Failed to allocate task");
00188
            LOG_END (" ");
00190
00191
00192
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
            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;
           pool->task_queue_tail = new_task;
00207
00208
00209
       pthread_cond_signal (&pool->queue_cond);
00210
        pthread_mutex_unlock (&pool->queue_lock);
LOG_END (" ");
00211
00212
00213 }
```

5.2.2.2 threadpool create()

return NULL;

00040

00041

5.2 ThreadPool 27

```
if (num_threads < 3)</pre>
00043
         {
00044
             num_threads = 3;
00045
        if (num_threads > NUM_CORES)
00046
        {
00047
            num_threads = NUM_CORES;
00049
00050
00051
        pool->num_threads = num_threads;
        pool->threads = malloc (sizeof (pthread_t) * num_threads);
00052
        if (pool->threads == NULL)
00053
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;
        pool->stop = FALSE;
pool->started = FALSE;
00063
00064
00065
00066
        if (pthread_mutex_init (&pool->queue_lock, NULL) != 0)
00067
00068
             perror ("Failed to initialize mutex");
00069
             free (pool->threads);
             free (pool);
LOG_END (" ");
00070
00071
00072
             return NULL:
00073
00074
00075
        if (pthread_cond_init (&pool->queue_cond, NULL) != 0)
00076
            perror ("Failed to initialize condition variable");
00077
             pthread_mutex_destroy (&pool->queue_lock);
free (pool->threads);
00078
00079
08000
             free (pool);
00081
             LOG_END (" ");
00082
             return NULL;
00083
          }
00084
00085
        if (pthread_cond_init (&pool->empty_cond, NULL) != 0)
00086
00087
            perror ("Failed to initialize empty condition variable");
00088
             pthread_mutex_destroy (&pool->queue_lock);
00089
             pthread_cond_destroy (&pool->queue_cond);
00090
            free (pool->threads);
free (pool);
LOG_END (" ");
00091
00092
00093
            return NULL;
00094
00095
        LOG_END (" ");
00096
00097
        return pool;
```

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()

```
void threadpool_destroy (
               struct threadpool * pool )
Definition at line 274 of file threadpool.c.
00275 {
00276
         LOG INIT (" ");
00277
         if (pool == NULL)
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
        pthread_mutex_destroy (&pool->queue_lock);
pthread_cond_destroy (&pool->queue_cond);
pthread_cond_destroy (&pool->empty_cond);
00294
00295
00296
```

5.2.2.5 threadpool empty()

free (pool->threads);
free (pool);
LOG_END (" ");

00297 00298

00299 00300 00301 }

```
int threadpool_empty ( {\tt struct\ threadpool\ *\ pool\ )}
```

Definition at line 240 of file threadpool.c.

```
00241 {
       LOG_INIT (" ");
00242
       if (pool == NULL)
00243
00244
           LOG_END (" ");
00245
           return TRUE;
00246
00247
00248
       pthread_mutex_lock (&pool->queue_lock);
00249
        int empty = (pool->task_queue_head == NULL);
00250
       pthread_mutex_unlock (&pool->queue_lock);
       LOG_END (" ");
00251
00252
       return empty;
00253 }
```

5.2.2.6 threadpool_init()

Definition at line 151 of file threadpool.c.

```
00152 {
00153   LOG_INIT (" ");
00154   if (pool == NULL || pool->started)
00155   {
    LOG_END (" ");
00157   return;
```

5.2 ThreadPool 29

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

5.2.2.7 threadpool_stop()

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

Definition at line 216 of file threadpool.c.

```
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
00235
       pool->started = FALSE;
00236
       LOG_END (" ");
```

5.2.2.8 threadpool_wait_empty()

Definition at line 256 of file threadpool.c.

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

30 Module Documentation

Chapter 6

Class Documentation

6.1 Node Struct Reference

Collaboration diagram for Node:

Public Attributes

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

6.1.1 Detailed Description

Definition at line 6 of file linked_list.c.

6.1.2 Member Data Documentation

6.1.2.1 children

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

Definition at line 10 of file linked_list.c.

6.1.2.2 next

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

Definition at line 11 of file linked_list.c.

6.1.2.3 ptr

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

Definition at line 8 of file linked_list.c.

6.1.2.4 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

6.2 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 []

6.2.1 Detailed Description

Definition at line 32 of file low_saurion.c.

6.2.2 Member Data Documentation

6.2.2.1 client_socket

```
int request::client_socket
```

Definition at line 41 of file low_saurion.c.

6.2.2.2 event_type

```
\verb|int request::event_type|\\
```

Definition at line 39 of file low_saurion.c.

6.2.2.3 iov

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

Definition at line 42 of file low_saurion.c.

6.2.2.4 iovec_count

```
size_t request::iovec_count
```

Definition at line 40 of file low_saurion.c.

6.2.2.5 next_iov

```
size_t request::next_iov
```

Definition at line 37 of file low_saurion.c.

6.2.2.6 next offset

```
size_t request::next_offset
```

Definition at line 38 of file low_saurion.c.

6.2.2.7 prev

void* request::prev

Definition at line 34 of file low_saurion.c.

6.2.2.8 prev_remain

```
size_t request::prev_remain
```

Definition at line 36 of file low_saurion.c.

6.2.2.9 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

6.3 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

6.3.1 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.

6.3.2 Member Data Documentation

6.3.2.1 efds

int* saurion::efds

Eventfd descriptors used for internal signaling between threads.

Definition at line 157 of file low_saurion.h.

6.3.2.2 list

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

Linked list for storing active requests.

Definition at line 159 of file low_saurion.h.

6.3.2.3 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.

6.3.2.4 n_threads

```
uint32_t saurion::n_threads
```

Number of threads in the thread pool.

Definition at line 169 of file low_saurion.h.

6.3.2.5 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.

6.3.2.6 pool

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

Thread pool for executing tasks in parallel.

Definition at line 167 of file low_saurion.h.

6.3.2.7 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.

6.3.2.8 ss

int saurion::ss

Server socket descriptor for accepting connections.

Definition at line 155 of file low_saurion.h.

6.3.2.9 status

int saurion::status

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

Definition at line 165 of file low_saurion.h.

6.3.2.10 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.

6.3.2.11 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

6.4 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

6.4.1 Detailed Description

Definition at line 7 of file saurion.hpp.

6.4.2 Member Typedef Documentation

6.4.2.1 ClosedCb

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

Definition at line 14 of file saurion.hpp.

6.4.2.2 ConnectedCb

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

Definition at line 10 of file saurion.hpp.

6.4.2.3 ErrorCb

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

Definition at line 15 of file saurion.hpp.

6.4.2.4 ReadedCb

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

Definition at line 11 of file saurion.hpp.

6.4.2.5 WroteCb

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

Definition at line 13 of file saurion.hpp.

6.4.3 Constructor & Destructor Documentation

6.4.3.1 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 }
```

6.4.3.2 ~Saurion()

```
Saurion::\simSaurion ( )
```

Definition at line 15 of file saurion.cpp.

```
00015 { saurion_destroy (this->s); }
```

6.4.3.3 Saurion() [2/3]

6.4.3.4 Saurion() [3/3]

6.4.4 Member Function Documentation

6.4.4.1 init()

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

Definition at line 18 of file saurion.cpp.

00019 {
00020         if (!saurion_start (this->s))
00021         {
00022             return;
00023         }
00024 }
```

6.4.4.2 on_closed()

Definition at line 57 of file saurion.cpp.

6.4.4.3 on_connected()

Definition at line 33 of file saurion.cpp.

6.4.4.4 on_error()

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 }
```

6.4.4.5 on_readed()

Definition at line 41 of file saurion.cpp.

6.4.4.6 on_wrote()

Definition at line 49 of file saurion.cpp.

6.4.4.7 operator=() [1/2]

6.4.4.8 operator=() [2/2]

6.4.4.9 send()

Definition at line 73 of file saurion.cpp.

```
00074 {
00075    saurion_send (this->s, fd, msg);
00076 }
```

6.4.4.10 stop()

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

Definition at line 27 of file saurion.cpp.
00028 {
00029     saurion_stop (this->s);
00030 }
```

6.4.5 Member Data Documentation

6.4.5.1 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

6.5 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

6.5.1 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.

6.5.2 Member Data Documentation

6.5.2.1 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.

6.5.2.2 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.

6.5.2.3 on_connected

void(* saurion::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 189 of file low_saurion.h.

6.5.2.4 on_connected_arg

void* saurion::saurion_callbacks::on_connected_arg

Additional argument for the connection callback.

Definition at line 191 of file low_saurion.h.

6.5.2.5 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.

Definition at line 233 of file low saurion.h.

6.5.2.6 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.

6.5.2.7 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.

6.5.2.8 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.

6.5.2.9 on_wrote

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

Callback for handling write events.

Parameters

f	d	File descriptor of the socket.
â	arg	Additional user-provided argument.

Definition at line 212 of file low_saurion.h.

6.5.2.10 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

6.6 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 31 of file low saurion.h.

6.6.2 Member Data Documentation

6.6.2.1 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.

6.6.2.2 on_closed_arg

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

Additional argument for the close callback.

Definition at line 73 of file low_saurion.h.

6.6.2.3 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.

Definition at line 39 of file low_saurion.h.

6.6.2.4 on_connected_arg

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

Additional argument for the connection callback.

Definition at line 41 of file low_saurion.h.

6.6.2.5 on_error

```
void(* saurion\_callbacks::on\_error) (const int fd, const char *const content, const ssize_\leftarrow t len, void *arg)
```

Callback for handling error events.

Parameters

fd	File descriptor of the socket where the error occurred.
	Pointer to the error message.
Generated by IEN	Doxygen Length of the error message.
arg	Additional user-provided argument.

Definition at line 83 of file low_saurion.h.

6.6.2.6 on_error_arg

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

Additional argument for the error callback.

Definition at line 86 of file low_saurion.h.

6.6.2.7 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.

6.6.2.8 on readed arg

```
void* saurion_callbacks::on_readed_arg
```

Additional argument for the read callback.

Definition at line 54 of file low_saurion.h.

6.6.2.9 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.

6.6.2.10 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

6.7 saurion_wrapper Struct Reference

Collaboration diagram for saurion_wrapper:

Public Attributes

- struct saurion * s
- uint32_t sel

6.7.1 Detailed Description

Definition at line 51 of file low_saurion.c.

6.7.2 Member Data Documentation

6.7.2.1 s

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

Definition at line 53 of file low_saurion.c.

6.7.2.2 sel

```
uint32_t saurion_wrapper::sel
```

Definition at line 54 of file low_saurion.c.

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

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

6.8 task Struct Reference

Collaboration diagram for task:

Public Attributes

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

6.8.1 Detailed Description

Definition at line 11 of file threadpool.c.

6.8.2 Member Data Documentation

6.8.2.1 argument

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

Definition at line 14 of file threadpool.c.

6.8.2.2 function

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

Definition at line 13 of file threadpool.c.

6.8.2.3 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

6.9 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

6.9.1 Detailed Description

Definition at line 18 of file threadpool.c.

6.9.2 Member Data Documentation

6.9.2.1 empty_cond

```
pthread_cond_t threadpool::empty_cond
```

Definition at line 26 of file threadpool.c.

6.9.2.2 num_threads

```
size_t threadpool::num_threads
```

Definition at line 21 of file threadpool.c.

6.9.2.3 queue_cond

```
\verb|pthread_cond_t| threadpool::queue\_cond|
```

Definition at line 25 of file threadpool.c.

6.9.2.4 queue_lock

```
pthread_mutex_t threadpool::queue_lock
```

Definition at line 24 of file threadpool.c.

6.9.2.5 started

int threadpool::started

Definition at line 28 of file threadpool.c.

6.9.2.6 stop

```
int threadpool::stop
```

Definition at line 27 of file threadpool.c.

6.9.2.7 task_queue_head

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

Definition at line 22 of file threadpool.c.

6.9.2.8 task_queue_tail

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

Definition at line 23 of file threadpool.c.

6.9.2.9 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
```

54 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 55

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.

56 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

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

Callback for handling socket closures.

58 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.

60 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.

62 File Documentation

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);
            void *on error arg:
00236
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
00326
```

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

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

```
#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:

64 File Documentation

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()

00094

00095

for (size_t i = 0; i < current->size; ++i)

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 69

```
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

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 lovec (struct lovec *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)
- 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 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

- pthread_mutex_t print_mutex
- 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

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()

Definition at line 250 of file low_saurion.c.

```
00253
        int res = ERROR_CODE;
00254
        pthread_mutex_lock (&s->m_rings[0]);
00255
        while (res != SUCCESS_CODE)
00256
          {
00257
            struct io_uring_sqe *sqe = io_uring_get_sqe (&s->rings[0]);
00258
            while (!sqe)
00259
              {
00260
                 sqe = io_uring_get_sqe (&s->rings[0]);
00261
                 nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00262
            struct request *req = NULL;
00263
00264
            if (!set_request (&req, &s->list, 0, NULL, 0))
00265
00266
                 free (sqe);
                 nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
res = ERROR_CODE;
00267
00268
00269
                 continue;
00270
00271
            req->client_socket = 0;
00272
            req->event_type = EV_ACC;
00273
             io_uring_prep_accept (sqe, s->ss, (struct sockaddr *const)ca, cal, 0);
00274
             io_uring_sqe_set_data (sqe, req);
00275
            if (io_uring_submit (&s->rings[0]) < 0)</pre>
00276
              {
00277
                 free (sqe);
                 list_delete_node (&s->list, req);
nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00278
00279
                 res = ERROR_CODE;
00280
00281
                 continue;
00282
00283
            res = SUCCESS_CODE;
00284
00285
       pthread_mutex_unlock (&s->m_rings[0]);
00286 }
```

7.13.2.2 add_efd()

```
static void add_efd (
              struct saurion *const s,
              const int client_socket,
              int sel ) [static]
Definition at line 327 of file low_saurion.c.
00328 {
```

add_fd (s, client_socket, sel);

7.13.2.3 add_fd()

00330 }

```
static void add_fd (
            struct saurion *const s,
             int client_socket,
            int sel ) [static]
```

Definition at line 289 of file low_saurion.c.

```
00290 {
00291
        int res = ERROR_CODE;
00292
        pthread_mutex_lock (&s->m_rings[sel]);
00293
        while (res != SUCCESS_CODE)
00294
00295
            struct io_uring *ring = &s->rings[sel];
00296
            struct io_uring_sqe *sqe = io_uring_get_sqe (ring);
00297
            while (!sqe)
00298
             {
                sqe = io_uring_get_sqe (ring);
nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00299
00300
00301
00302
            struct request *req = NULL;
00303
            if (!set_request (&req, &s->list, CHUNK_SZ, NULL, 0))
00304
00305
                free (sqe);
00306
                res = ERROR_CODE;
00307
                continue;
00308
00309
             req->event_type = EV_REA;
00310
             req->client_socket = client_socket;
00311
            io_uring_prep_readv (sqe, client_socket, &req->iov[0], req->iovec_count,
00312
                                  0):
00313
            io_uring_sqe_set_data (sqe, req);
00314
            if (io_uring_submit (ring) < 0)</pre>
00315
                free (sqe);
00316
00317
                list_delete_node (&s->list, req);
00318
                 res = ERROR_CODE;
00319
                continue;
00320
00321
            res = SUCCESS_CODE;
00323 pthread_mutex_unlock (&s->m_rings[sel]);
00324 }
00322
```

7.13.2.4 add_read()

```
static void add_read (
            struct saurion *const s,
            const int client_socket ) [static]
```

Definition at line 333 of file low saurion.c.

```
00334 {
       int sel = next (s);
00335
00336
       add_fd (s, client_socket, sel);
00337 }
```

7.13.2.5 add_read_continue()

```
static void add_read_continue (
              struct saurion *const s,
              struct request * oreq,
              const int sel ) [static]
Definition at line 340 of file low_saurion.c.
00342 {
00343
        pthread_mutex_lock (&s->m_rings[sel]);
00344
       int res = ERROR_CODE;
00345
       while (res != SUCCESS_CODE)
00346
         {
00347
           struct io_uring *ring = &s->rings[sel];
00348
           struct io_uring_sqe *sqe = io_uring_get_sqe (ring);
00349
           while (!sqe)
00350
             {
00351
               sqe = io_uring_get_sqe (ring);
00352
               nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00353
00354
            if (!set_request (&oreq, &s->list, oreq->prev_remain, NULL, 0))
00355
             {
00356
               free (sqe);
00357
                res = ERROR_CODE;
00358
               continue;
00359
00360
            io_uring_prep_readv (sqe, oreq->client_socket, &oreq->iov[0],
00361
                                 oreq->iovec_count, 0);
            io_uring_sqe_set_data (sqe, oreq);
00362
           if (io_uring_submit (ring) < 0)
00363
00364
             {
00365
                free (sqe);
00366
                list_delete_node (&s->list, oreq);
00367
                res = ERROR_CODE;
00368
               continue;
00369
00370
           res = SUCCESS_CODE;
00371
00372
       pthread_mutex_unlock (&s->m_rings[sel]);
00373 }
```

7.13.2.6 add_write()

Definition at line 376 of file low_saurion.c.

```
00378 {
00379
        int res = ERROR_CODE;
00380
        pthread_mutex_lock (&s->m_rings[sel]);
00381
        while (res != SUCCESS_CODE)
00382
00383
            struct io_uring *ring = &s->rings[sel];
00384
            struct io_uring_sqe *sqe = io_uring_get_sqe (ring);
00385
            while (!sge)
00386
             {
00387
                sqe = io_uring_get_sqe (ring);
00388
                nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00389
00390
            struct request *req = NULL;
            if (!set_request (&req, &s->list, strlen (str), (const void *const)str,
00391
00392
                              1))
00393
00394
                free (sqe);
00395
                res = ERROR_CODE;
00396
                continue;
00397
              }
00398
            req->event_type = EV_WRI;
00399
            req->client_socket = fd;
```

```
io_uring_prep_writev (sqe, req->client_socket, req->iov,
00401
                                 req->iovec_count, 0);
           io_uring_sqe_set_data (sqe, req);
00402
00403
           if (io_uring_submit (ring) < 0)</pre>
00404
00405
                free (sqe);
00406
                list_delete_node (&s->list, req);
00407
                res = ERROR_CODE;
00408
               nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00409
               continue;
             }
00410
00411
           res = SUCCESS_CODE;
00412
00413
       pthread_mutex_unlock (&s->m_rings[sel]);
00414 }
```

7.13.2.7 handle_accept()

```
static void handle_accept (  {\tt const\ struct\ saurion\ *const\ } s, \\ {\tt const\ int\ } fd\ ) \ \ [static]
```

Definition at line 418 of file low_saurion.c.

7.13.2.8 handle_close()

Definition at line 659 of file low_saurion.c.

7.13.2.9 handle_error()

Definition at line 648 of file low_saurion.c.

7.13.2.10 handle_read()

```
static void handle_read (
              struct saurion *const s,
              struct request *const req ) [static]
Definition at line 601 of file low_saurion.c.
00602 {
        void *msg = NULL;
00603
       size_t len = 0;
while (1)
00604
00605
00606
         {
00607
            if (!read_chunk (&msg, &len, req))
00608
              {
               break;
00609
00610
00611
            if (req->next_iov || req->next_offset)
00612
00613
                if (s->cb.on_readed && msg)
00614
00615
                   s->cb.on_readed (req->client_socket, msg, len,
00616
                                    s->cb.on_readed_arg);
00617
00618
               free (msg);
00619
               msg = NULL;
00620
                continue;
00621
            if (req->prev && req->prev_size && req->prev_remain)
00622
            {
00623
00624
               add read continue (s, reg, next (s));
00625
               return;
00626
00627
            if (s->cb.on_readed && msg)
00628
            {
               s->cb.on_readed (req->client_socket, msg, len, s->cb.on_readed_arg);
00629
00630
00631
           free (msg);
00632
           msg = NULL;
00633
           break;
00634
       add_read (s, req->client_socket);
00635
00636 }
```

7.13.2.11 handle_write()

Definition at line 639 of file low_saurion.c.

7.13.2.12 htonll()

Definition at line 65 of file low saurion.c.

```
00066 { 00067 int num = 42;
```

7.13.2.13 next()

```
static uint32_t next ( struct saurion * s ) [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 }
```

7.13.2.14 ntohll()

Definition at line 78 of file low_saurion.c.

7.13.2.15 saurion_worker_master()

Definition at line 891 of file low_saurion.c.

```
00892 {
        LOG_INIT (" ");
00893
        struct saurion *const s = (struct saurion *)arg; struct sockaddr_in client_addr;
00894
00895
00896
        socklen_t client_addr_len = sizeof (client_addr);
00897
00898
        add_efd (s, s->efds[0], 0);
00899
       add_accept (s, &client_addr, &client_addr_len);
00900
00901
        pthread_mutex_lock (&s->status_m);
        ++s->status;
00902
00903
        pthread_cond_broadcast (&s->status_c);
00904
        pthread_mutex_unlock (&s->status_m);
00905
        while (1)
00906
         {
00907
00908
                = saurion_worker_master_loop_it (s, &client_addr, &client_addr_len);
```

```
if (ret == ERROR_CODE || ret == CRITICAL_CODE)
00910
00911
               break;
             }
00912
00913
00914
      pthread_mutex_lock (&s->status_m);
        --s->status;
00916
       pthread_cond_signal (&s->status_c);
00917
       pthread_mutex_unlock (&s->status_m);
00918
       LOG_END (" ");
00919
       return;
00920 }
```

7.13.2.16 saurion worker master loop it()

Definition at line 823 of file low_saurion.c.

```
00826 {
00827
         LOG INIT (" ");
         struct io_uring ring = s->rings[0];
struct io_uring_cqe *cqe = NULL;
int ret = io_uring_wait_cqe (&ring, &cqe);
00828
00829
00830
         if (ret < 0)
00832
             free (cqe);
LOG_END (" ");
00833
00834
             return CRITICAL_CODE;
00835
00836
00837
         struct request *req = (struct request *)cqe->user_data;
00838
        if (!req)
00839
            io_uring_cqe_seen (&s->rings[0], cqe);
LOG_END (" ");
00840
00841
             return SUCCESS_CODE;
00842
00843
00844
         if (cqe->res < 0)</pre>
00845
         {
             list_delete_node (&s->list, req);
LOG_END (" ");
00846
00847
00848
             return CRITICAL_CODE;
00849
00850
            (req->client_socket == s->efds[0])
00851
00852
             io_uring_cqe_seen (&s->rings[0], cqe);
00853
             list_delete_node (&s->list, req);
LOG_END (" ");
00854
00855
             return ERROR_CODE;
00856
00857
        io_uring_cqe_seen (&s->rings[0], cqe);
00858
        switch (req->event_type)
00859
           case EV_ACC:
00860
00861
           handle_accept (s, cqe->res);
             add_accept (s, client_addr, client_addr_len);
00862
00863
             add_read (s, cqe->res);
00864
             list_delete_node (&s->list, req);
           break;
case EV_REA:
00865
00866
00867
            if (cqe->res < 0)</pre>
00868
              {
00869
                  handle_error (s, req);
00870
00871
             if (cqe->res < 1)</pre>
00872
             {
00873
                 handle_close (s, req);
00874
00875
             if (cqe->res > 0)
00876
               {
00877
                 handle_read (s, req);
00878
00879
             list_delete_node (&s->list, req);
00880
             break;
           case EV_WRI:
```

7.13.2.17 saurion worker slave()

```
void saurion_worker_slave (
     void * arg )
```

Definition at line 986 of file low saurion.c.

```
00987 {
00988
        LOG INIT (" ");
        struct saurion_wrapper *const ss = (struct saurion_wrapper *)arg;
00989
00990
        struct saurion *s = ss->s;
00991
        const int sel = ss->sel;
00992
       free (ss);
00993
00994
       add efd (s, s->efds[sel], sel);
00995
00996
       pthread_mutex_lock (&s->status_m);
00997
00998
        pthread_cond_broadcast (&s->status_c);
00999
        pthread_mutex_unlock (&s->status_m);
01000
        while (1)
01001
01002
            int res = saurion_worker_slave_loop_it (s, sel);
01003
            if (res == ERROR_CODE || res == CRITICAL_CODE)
01004
             {
01005
                break:
              }
01006
01007
01008
       pthread_mutex_lock (&s->status_m);
01009
         -s->status;
01010
        pthread_cond_signal (&s->status_c);
       pthread_mutex_unlock (&s->status_m);
LOG_END (" ");
01011
01012
01013
        return;
01014 }
```

7.13.2.18 saurion_worker_slave_loop_it()

```
static int saurion_worker_slave_loop_it (  struct \ saurion *const \ s,   const \ int \ sel \ ) \ \ [static]
```

Definition at line 924 of file low_saurion.c.

```
00925 {
         LOG_INIT (" ");
00926
        struct io_uring ring = s->rings[sel];
struct io_uring_cqe *cqe = NULL;
00927
00928
00929
00930
         add_efd (s, s->efds[sel], sel);
00931
         int ret = io_uring_wait_cqe (&ring, &cqe);
00932
         if (ret < 0)</pre>
00933
             free (cqe);
LOG_END (" ");
00934
00935
             return CRITICAL_CODE;
00936
00937
00938
         struct request *req = (struct request *)cqe->user_data;
00939
        if (!req)
00940
00941
             io_uring_cqe_seen (&ring, cqe);
00942
             LOG_END (" ");
00943
             return SUCCESS_CODE;
```

```
00944
00945
        if (cqe->res < 0)</pre>
00946
           list_delete_node (&s->list, req);
LOG_END (" ");
00947
00948
           return CRITICAL_CODE;
00949
00950
00951
        if (req->client_socket == s->efds[sel])
00952
00953
            io_uring_cqe_seen (&ring, cqe);
          list_delete_node (&s->list, req);
LOG_END (" ");
00954
00955
00956
            return ERROR_CODE;
00957
00958
       io_uring_cqe_seen (&ring, cqe);
00959
       switch (req->event_type)
00960
00961
         case EV REA:
00962
           if (cqe->res < 0)</pre>
            {
00963
00964
                handle_error (s, req);
00965
00966
            if (cqe->res < 1)</pre>
00967
00968
                handle_close (s, req);
00969
00970
            if (cqe->res > 0)
00971
00972
                handle_read (s, req);
00973
00974
           list_delete_node (&s->list, req);
00975
            break;
00976
        case EV_WRI:
00977
            handle_write (s, req->client_socket);
00978
            list_delete_node (&s->list, req);
00979
            break;
00980
00982 return SUCCESS_CODE;
00983 }
00981 LOG_END (" ");
```

7.13.3 Variable Documentation

7.13.3.1 print mutex

pthread_mutex_t print_mutex

Definition at line 47 of file low_saurion.c.

7.13.3.2 TIMEOUT_RETRY_SPEC

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

Definition at line 49 of file low_saurion.c.

7.14 low saurion.c

00081

00082

if (*(char *)&num == 42)

```
Go to the documentation of this file.
00001 #include "low_saurion.h" 00002 #include "config.h"
00002 #include "config.h" // for ERROR_CODE, SUCCESS_CODE, CHUNK_SZ
00003 #include "linked_list.h" // for list_delete_node, list_free, list_insert
00004 #include "threadpool.h" // for threadpool_add, threadpool_create
00006 #include <arpa/inet.h> // for hton1, ntoh1, htons
00007 #include <bits/socket-constants.h> // for SOL_SOCKET, SO_REUSEADDR
00007 #include <bits/socket-constants.h> // for SOL_SOCKET, SO_REUSEADDR
00008 #include <liburing.h> // for io_uring_get_sqe, io_uring, io_uring_...
00009 #include <liburing/io_uring.h> // for io_uring_cqe
0010 #include <nanologger.h> // for LOG_END, LOG_INIT
00011 #include <netinet/in.h> // for sockaddr_in, INADDR_ANY, in_addr
00012 #include <pthread.h> // for pthread_mutex_lock, pthread_mutex_unlock
00013 #include <stdint.h> // for uint32_t, uint64_t, uint8_t
00014 #include <stdio.h> // for NULL
00015 #include <stdib.h> // for free, malloc
00016 #include <string.h> // for memset, memcpy, strlen
00017 #include <sys/eventfd.h> // for eventfd, EFD_NONBLOCK
00018 #include <sys/socket.h> // for socklen_t, bind, listen, setsockopt
00019 #include <syy/uio.h> // for nanosleep
00020 #include <time.h>
                                                               // for nanosleep
// for close, write
00021 #include <unistd.h>
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 {
             void *prev;
00034
             size_t prev_size;
size_t prev_remain;
00035
00036
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 pthread_mutex_t print_mutex;
00049 struct timespec TIMEOUT_RETRY_SPEC = { 0, TIMEOUT_RETRY * 1000L };
00050
00051 struct saurion_wrapper
00052 {
00053 struct saurion *s;
             uint32_t sel;
00055 };
00056
00057 static uint32_t
00058 next (struct saurion *s)
00059 {
00060 s->next = (s->next + 1) % s->n_threads;
00061 return s->next;
00062 }
00063
00064 static uint64 t
00065 hton11 (uint64 t value)
00066 {
00067 int num = 42;
00068
              if (*(char *)&num == 42)
00069
                    uint32_t high_part = hton1 ((uint32_t) (value » 32));
00070
                uint32_t low_part = hton1 ((uint32_t)(value % 32));
uint32_t low_part = hton1 ((uint32_t)(value & 0xFFFFFFFLL));
00071
                    return ((uint64_t)low_part « 32) | high_part;
00072
00074 return value;
00075 }
00076
00077 static uint64_t
00078 ntohll (uint64_t value)
08000
             int num = 42;
```

7.14 low_saurion.c 83

```
uint32_t high_part = ntohl ((uint32_t) (value » 32));
00084
            uint32_t low_part = ntohl ((uint32_t) (value & 0xFFFFFFFFLL));
00085
            return ((uint64_t)low_part « 32) | high_part;
00086
00087
        return value;
00088 }
00090 void
00091 free_request (struct request *req, void **children_ptr, size_t amount)
00092 {
00093
        if (children_ptr)
00094
          {
             free (children_ptr);
00095
00096
            children_ptr = NULL;
00097
00098
        for (size_t i = 0; i < amount; ++i)</pre>
00099
            free (req->iov[i].iov_base);
00100
            req->iov[i].iov_base = NULL;
00101
00102
00103
        free (req);
       req = NULL;
free (children_ptr);
00104
00105
00106
       children_ptr = NULL;
00107 }
00108
00109 [[nodiscard]]
00110 int
00111 initialize_iovec (struct iovec *iov, size_t amount, size_t pos,
00112
                         const void *msg, size_t size, uint8_t h)
00113 {
00114
        if (!iov || !iov->iov_base)
00115
         {
00116
            return ERROR_CODE;
00117
        if (msg)
00118
00119
        {
            size_t len = iov->iov_len;
            char *dest = (char *)iov->iov_base;
char *orig = (char *)msg + pos * CHUNK_SZ;
00121
00122
00123
            size_t cpy_sz = 0;
            if (h)
00124
00125
              {
                 if (pos == 0)
00126
00127
                  {
00128
                     uint64_t send_size = htonll (size);
                     memcpy (dest, &send_size, sizeof (uint64_t));
dest += sizeof (uint64_t);
len -= sizeof (uint64_t);
00129
00130
00131
00132
00133
                 else
00134
00135
                     orig -= sizeof (uint64_t);
00136
                 if ((pos + 1) == amount)
00137
00138
                  {
--len;
00140
                     cpy_sz = (len < size ? len : size);</pre>
00141
                    dest[cpy_sz] = 0;
00142
                   }
00143
              }
            cpy_sz = (len < size ? len : size);
00144
            memcpy (dest, orig, cpy_sz);
dest += cpy_sz;
size_t rem = CHUNK_SZ - (dest - (char *)iov->iov_base);
00145
00146
00147
00148
            memset (dest, 0, rem);
00149
00150
        else
00151
        {
00152
            memset ((char *)iov->iov_base, 0, CHUNK_SZ);
00153
00154
        return SUCCESS_CODE;
00155 }
00156
00157 [[nodiscard]]
00159 allocate_iovec (struct iovec *iov, size_t amount, size_t pos, size_t size,
00160
                       void **chd_ptr)
00161 {
00162
        if (!iov || !chd_ptr)
00163
        {
00164
            return ERROR_CODE;
00165
00166
        iov->iov_base = malloc (CHUNK_SZ);
00167
        if (!iov->iov_base)
00168
00169
            return ERROR_CODE;
```

```
00171
        iov->iov_len = (pos == (amount - 1) ? (size % CHUNK_SZ) : CHUNK_SZ);
00172
        if (iov->iov_len == 0)
00173
            iov->iov_len = CHUNK SZ:
00174
00175
00176
        chd_ptr[pos] = iov->iov_base;
00177
        return SUCCESS_CODE;
00178 }
00179
00180 [[nodiscard]]
00181 int
00182 set_request (struct request **r, struct Node **1, size_t s, const void *m,
00183
                   uint8_t h)
00184 {
00185
        uint64_t full_size = s;
00186
        if (h)
00187
        {
            full_size += (sizeof (uint64_t) + sizeof (uint8_t));
00189
00190
        size_t amount = full_size / CHUNK_SZ;
        amount = amount + (full_size % CHUNK_SZ == 0 ? 0 : 1);
00191
        struct\ request\ *temp = (struct\ request\ *) malloc (
00192
00193
            sizeof (struct request) + sizeof (struct iovec) * amount);
00194
        if (!temp)
00195
        {
00196
            return ERROR_CODE;
00197
        <u>if</u> (!*r)
00198
        {
00199
00200
           *r = temp;
00201
            (*r)->prev = NULL;
00202
            (*r)->prev_size = 0;
00203
            (*r) \rightarrow prev\_remain = 0;
00204
            (*r) \rightarrow next_iov = 0;
            (*r)->next_offset = 0;
00205
00206
       else
00208
        {
00209
            temp->client_socket = (*r)->client_socket;
00210
            temp->event_type = (*r)->event_type;
temp->prev = (*r)->prev;
00211
            temp->prev_size = (*r)->prev_size;
temp->prev_remain = (*r)->prev_remain;
00212
00213
00214
            temp->next_iov = (*r)->next_iov;
00215
            temp->next_offset = (*r)->next_offset;
00216
            *r = temp;
00217
00218
       struct request *req = *r;
00219
        reg->iovec count = (int)amount;
00220
        void **children_ptr = (void **)malloc (amount * sizeof (void *));
00221
        if (!children_ptr)
00222
00223
            free_request (req, children_ptr, 0);
00224
            return ERROR_CODE;
00225
        for (size_t i = 0; i < amount; ++i)</pre>
00227
00228
            if (!allocate_iovec (&req->iov[i], amount, i, full_size, children_ptr))
00229
00230
                free_request (req, children_ptr, amount);
00231
                return ERROR CODE;
00232
00233
            if (!initialize_iovec (&req->iov[i], amount, i, m, s, h))
00234
00235
                free_request (req, children_ptr, amount);
00236
                return ERROR_CODE;
00237
00238
00239
        if (list_insert (l, req, amount, children_ptr))
00240
00241
            free_request (req, children_ptr, amount);
00242
            return ERROR_CODE;
00243
00244
       free (children_ptr);
       return SUCCESS_CODE;
00245
00246 }
00247
00248 /*********** ADDERS ************/
00249 static void
00250 add_accept (struct saurion *const s, struct sockaddr_in *const ca,
                  socklen_t *const cal)
00252 {
00253
       int res = ERROR_CODE;
00254
        pthread_mutex_lock (&s->m_rings[0]);
00255
        while (res != SUCCESS_CODE)
00256
         -{
```

7.14 low saurion.c 85

```
struct io_uring_sqe *sqe = io_uring_get_sqe (&s->rings[0]);
00258
            while (!sqe)
00259
00260
                 sqe = io_uring_get_sqe (&s->rings[0]);
00261
                 nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00262
            struct request *req = NULL;
00263
00264
            if (!set_request (&req, &s->list, 0, NULL, 0))
00265
00266
                free (sqe);
                 nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00267
                 res = ERROR_CODE;
00268
00269
                continue;
00270
00271
            req->client_socket = 0;
00272
             req->event_type = EV_ACC;
00273
             io_uring_prep_accept (sqe, s->ss, (struct sockaddr *const)ca, cal, 0);
00274
            io_uring_sqe_set_data (sqe, req);
            if (io_uring_submit (&s->rings[0]) < 0)</pre>
00276
              {
00277
                 free (sqe);
                 list_delete_node (&s->list, req);
nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00278
00279
                 res = ERROR_CODE;
00280
00281
                continue;
00283
            res = SUCCESS_CODE;
00284
00285
       pthread_mutex_unlock (&s->m_rings[0]);
00286 }
00287
00288 static void
00289 add_fd (struct saurion *const s, int client_socket, int sel)
00290 {
00291
        int res = ERROR_CODE;
        pthread_mutex_lock (&s->m_rings[sel]);
while (res != SUCCESS_CODE)
00292
00293
00294
00295
            struct io_uring *ring = &s->rings[sel];
00296
            struct io_uring_sqe *sqe = io_uring_get_sqe (ring);
00297
            while (!sqe)
00298
              {
                sqe = io_uring_get_sqe (ring);
nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00299
00300
00301
00302
             struct request *req = NULL;
00303
             if (!set_request (&req, &s->list, CHUNK_SZ, NULL, 0))
00304
              {
00305
                 free (sge);
00306
                res = ERROR_CODE;
00307
                continue;
00308
00309
             req->event_type = EV_REA;
00310
             req->client_socket = client_socket;
            io_uring_prep_readv (sqe, client_socket, &req->iov[0], req->iovec_count,
00311
00312
                                   0);
00313
            io_uring_sqe_set_data (sqe, req);
00314
            if (io_uring_submit (ring) < 0)</pre>
00315
00316
                 free (sqe);
                list_delete_node (&s->list, req);
res = ERROR_CODE;
00317
00318
00319
                continue;
00320
00321
            res = SUCCESS_CODE;
00322
00323
        pthread_mutex_unlock (&s->m_rings[sel]);
00324 }
00325
00326 static void
00327 add_efd (struct saurion *const s, const int client_socket, int sel)
00328 {
00329
        add_fd (s, client_socket, sel);
00330 }
00331
00332 static void
00333 add_read (struct saurion *const s, const int client_socket)
00334 {
00335
        int sel = next (s);
00336
       add_fd (s, client_socket, sel);
00337 }
00338
00339 static void
00340 add_read_continue (struct saurion *const s, struct request *oreq,
00341
                          const int sel)
00342 {
00343
       pthread mutex lock (&s->m rings[sell);
```

```
int res = ERROR_CODE;
00345
        while (res != SUCCESS_CODE)
00346
00347
            struct io_uring *ring = &s->rings[sel];
00348
            struct io_uring_sqe *sqe = io_uring_get_sqe (ring);
00349
            while (!sqe)
00350
             {
00351
                sqe = io_uring_get_sqe (ring);
00352
                nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00353
00354
            if (!set_request (&oreq, &s->list, oreq->prev_remain, NULL, 0))
00355
              {
00356
                free (sqe);
                res = ERROR_CODE;
00357
00358
                continue;
00359
            io_uring_prep_readv (sqe, oreq->client_socket, &oreq->iov[0],
00360
00361
                                 oreq->iovec_count, 0);
            io_uring_sqe_set_data (sqe, oreq);
00362
00363
            if (io_uring_submit (ring) < 0)</pre>
00364
00365
                free (sqe);
                list_delete_node (&s->list, oreq);
00366
00367
                res = ERROR CODE;
00368
                continue;
00369
00370
            res = SUCCESS_CODE;
00371
00372
       pthread_mutex_unlock (&s->m_rings[sel]);
00373 }
00374
00375 static void
00376 add_write (struct saurion *const s, int fd, const char *const str,
00377
                 const int sel)
00378 {
       int res = ERROR_CODE;
00379
       pthread_mutex_lock (&s->m_rings[sel]);
while (res != SUCCESS_CODE)
00380
00382
         {
00383
            struct io_uring *ring = &s->rings[sel];
00384
            struct io_uring_sqe *sqe = io_uring_get_sqe (ring);
00385
            while (!sge)
00386
             {
00387
                sqe = io_uring_get_sqe (ring);
00388
                nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00389
00390
            struct request *req = NULL;
            if (!set_request (&req, &s->list, strlen (str), (const void *const)str,
00391
00392
                               1))
00393
00394
                free (sqe);
00395
                res = ERROR_CODE;
00396
                continue;
00397
            req->event_type = EV_WRI;
req->client_socket = fd;
00398
00399
00400
            io_uring_prep_writev (sqe, req->client_socket, req->iov,
00401
                                   req->iovec_count, 0);
00402
            io_uring_sqe_set_data (sqe, req);
00403
            if (io_uring_submit (ring) < 0)</pre>
00404
              {
00405
                free (sqe);
00406
                list_delete_node (&s->list, req);
00407
                res = ERROR_CODE;
00408
                nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00409
                continue;
00410
            res = SUCCESS_CODE;
00411
00412
00413
       pthread_mutex_unlock (&s->m_rings[sel]);
00414 }
00415
00416 /************** HANDLERS *************
00417 static void
00418 handle accept (const struct saurion *const s, const int fd)
00419 {
00420
        if (s->cb.on_connected)
00421
00422
            s->cb.on_connected (fd, s->cb.on_connected_arg);
00423
          }
00424 }
00425
00426 [[nodiscard]]
00427 int
00428 read_chunk (void **dest, size_t *len, struct request *const req)
00429 {
00430
       if (reg->iovec count == 0)
```

7.14 low_saurion.c 87

```
{
00432
            return ERROR_CODE;
00433
00434
        size_t max_iov_cont = 0; //< Total size of request</pre>
00435
00436
        for (size_t i = 0; i < req->iovec_count; ++i)
00437
00438
            max_iov_cont += req->iov[i].iov_len;
00439
00440
        size_t cont_sz = 0;
        size_t cont_rem = 0;
00441
        size_t curr_iov = 0;
00442
00443
        size_t curr_iov_off = 0;
00444
        size_t dest_off = 0;
00445
        void *dest_ptr = NULL;
00446
        if (req->prev && req->prev_size && req->prev_remain)
00447
00448
            cont_sz = req->prev_size;
            cont_rem = req->prev_remain;
00449
00450
            curr_iov = 0;
00451
             curr_iov_off = 0;
00452
             dest_off = cont_sz - cont_rem;
00453
            if (cont_rem <= max_iov_cont)</pre>
00454
              {
00455
                 *dest = req->prev;
00456
                dest_ptr = *dest;
00457
                 req->prev = NULL;
00458
                 req->prev_size = 0;
00459
                 req->prev_remain = 0;
00460
              }
00461
            else
00462
              {
00463
                 dest_ptr = req->prev;
00464
                 *dest = NULL;
00465
00466
00467
        else if (req->next iov || req->next offset)
00468
00469
            curr_iov = req->next_iov;
00470
            curr_iov_off = req->next_offset;
00471
            cont_sz = *(
            (size_t *)(((uint8_t *)req->iov[curr_iov].iov_base) + curr_iov_off));
cont_sz = ntohll (cont_sz);
00472
00473
00474
            curr_iov_off += sizeof (uint64_t);
            cont_rem = cont_sz;
dest_off = cont_sz - cont_rem;
00475
00476
00477
             if ((curr_iov_off + cont_rem + 1) <= max_iov_cont)</pre>
00478
00479
                 *dest = malloc (cont_sz);
00480
                 dest_ptr = *dest;
00481
00482
00483
              {
                req->prev = malloc (cont_sz);
dest_ptr = req->prev;
00484
00485
                *dest = NULL;
*len = 0;
00486
00487
00488
00489
00490
        else
00491
           curr_iov = 0;
00492
00493
            curr_iov_off = 0;
00494
            cont_sz = *(
00495
                 (size_t *)(((uint8_t *)req->iov[curr_iov].iov_base) + curr_iov_off));
00496
             cont_sz = ntohll (cont_sz);
00497
            curr_iov_off += sizeof (uint64_t);
00498
            cont_rem = cont_sz;
            dest_off = cont_sz - cont_rem;
00499
            if (cont_rem <= max_iov_cont)</pre>
00501
00502
                 *dest = malloc (cont_sz);
00503
                 dest_ptr = *dest;
00504
00505
            else
00506
              {
00507
                 req->prev = malloc (cont_sz);
00508
                 dest_ptr = req->prev;
00509
                 *dest = NULL:
00510
              }
00511
00512
        size_t curr_iov_msg_rem = 0;
00513
00514
        uint8_t ok = 1UL;
00515
        while (1)
00516
00517
            curr iov msg rem
```

```
= MIN (cont_rem, (req->iov[curr_iov].iov_len - curr_iov_off));
           00519
00520
                   curr_iov_msg_rem);
00521
            dest_off += curr_iov_msg_rem;
00522
00523
            curr_iov_off += curr_iov_msq_rem;
            cont_rem -= curr_iov_msg_rem;
00525
            if (cont_rem <= \overline{0})
00526
00527
                if (*(((uint8_t *)req->iov[curr_iov].iov_base) + curr_iov_off) != 0)
00528
                 {
00529
                   ok = OUL;
00530
00531
                *len = cont_sz;
00532
                ++curr_iov_off;
00533
               break;
00534
00535
            if (curr iov off >= (req->iov[curr iov].iov len))
00536
00537
                ++curr_iov;
00538
                if (curr_iov == req->iovec_count)
00539
00540
                   break;
00541
00542
                curr_iov_off = 0;
00543
00544
         }
00545
00546
        if (req->prev)
00547
        {
           req->prev_size = cont_sz;
req->prev_remain = cont_rem;
00548
00549
00550
            *dest = NULL;
00551
            len = 0;
00552
00553
        else
00554
        {
           req->prev_size = 0;
00556
            req->prev_remain = 0;
00557
00558
        if (curr_iov < req->iovec_count)
00559
00560
            uint64_t next_sz = *(uint64_t *)(((uint8_t *)req->iov[curr_iov].iov_base)
00561
                                              + curr_iov_off);
00562
            if ((req->iov[curr_iov].iov_len > curr_iov_off) && next_sz)
00563
00564
                req->next_iov = curr_iov;
00565
                req->next_offset = curr_iov_off;
00566
00567
            else
00568
             {
00569
                req->next_iov = 0;
00570
                req->next_offset = 0;
00571
00572
         }
00573
00574
        if (ok)
00575
        {
00576
           return SUCCESS_CODE;
00577
00578
       free (dest_ptr);
00579
       dest_ptr = NULL;
        *dest = NULL;
*len = 0;
00580
00581
00582
        req->next_iov = 0;
        req->next_offset = 0;
00583
        for (size_t i = curr_iov; i < req->iovec_count; ++i)
00584
00585
00586
            for (size_t j = curr_iov_off; j < req->iov[i].iov_len; ++j)
00588
                uint8_t foot = *((uint8_t *)req->iov[i].iov_base) + j;
00589
                if (foot == 0)
00590
                 {
                   req->next_iov = i;
req->next_offset = (j + 1) % req->iov[i].iov_len;
00591
00592
00593
                    return ERROR_CODE;
00594
00595
00596
00597
       return ERROR CODE:
00598 }
00600 static void
00601 handle_read (struct saurion *const s, struct request *const req)
00602 {
       void *msg = NULL;
00603
00604
       size_t len = 0;
```

7.14 low_saurion.c 89

```
00605
       while (1)
00606
00607
            if (!read_chunk (&msg, &len, req))
00608
             {
00609
               break:
00610
            if (req->next_iov || req->next_offset)
00611
00612
              {
00613
                if (s->cb.on_readed && msg)
00614
00615
                   s->cb.on_readed (req->client_socket, msg, len,
00616
                                    s->cb.on_readed_arg);
00617
00618
               free (msg);
00619
               msg = NULL;
               continue;
00620
00621
00622
           if (req->prev && req->prev_size && req->prev_remain)
00623
00624
               add_read_continue (s, req, next (s));
00625
00626
           if (s->cb.on_readed && msg)
00627
00628
00629
               s->cb.on_readed (req->client_socket, msg, len, s->cb.on_readed_arg);
00630
00631
           free (msg);
00632
           msg = NULL;
00633
           break;
00634
00635
       add_read (s, req->client_socket);
00636 }
00637
00638 static void
00639 handle_write (const struct saurion *const s, const int fd)
00640 {
00641
       if (s->cb.on wrote)
00642
00643
           s->cb.on_wrote (fd, s->cb.on_wrote_arg);
00644
00645 }
00646
00647 static void
00648 handle_error (const struct saurion *const s, const struct request *const req)
00649 {
00650
        if (s->cb.on_error)
00651
           const char *resp = "ERROR";
00652
           s->cb.on_error (req->client_socket, resp, (ssize_t)strlen (resp),
00653
00654
                           s->cb.on error arg);
00655
         }
00656 }
00657
00658 static void
00659 handle_close (const struct saurion *const s, const struct request *const req)
00660 {
00661
       if (s->cb.on_closed)
       {
00662
00663
           s->cb.on_closed (req->client_socket, s->cb.on_closed_arg);
00664
00665
       close (req->client_socket);
00666 }
00667
00669 int
00670 saurion_set_socket (const int p)
00671 {
00672
       int sock = 0;
00673
       struct sockaddr in srv addr:
00675
       sock = socket (PF_INET, SOCK_STREAM, 0);
00676
       if (sock < 1)
00677
        {
00678
           return ERROR CODE:
00679
         }
00680
00681
00682
       if (setsockopt (sock, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof (int)) < 0)</pre>
00683
           return ERROR_CODE;
00684
00685
00686
00687
       memset (&srv_addr, 0, sizeof (srv_addr));
00688
        srv_addr.sin_family = AF_INET;
       srv_addr.sin_port = htons (p);
srv_addr.sin_addr.s_addr = hton1 (INADDR_ANY);
00689
00690
00691
```

```
if (bind (sock, (const struct sockaddr *)&srv_addr, sizeof (srv_addr)) < 0)</pre>
00693
00694
            return ERROR_CODE;
         }
00695
00696
00697
        if (listen (sock, ACCEPT_QUEUE) < 0)</pre>
00699
            return ERROR_CODE;
00700
00701
00702
        return sock;
00703 }
00704
00705 [[nodiscard]]
00706 struct saurion *
00707 saurion_create (uint32_t n_threads)
00708 {
00709
        LOG INIT (" ");
        struct saurion *p = (struct saurion *)malloc (sizeof (struct saurion));
00710
00711
        if (!p)
00712
            LOG_END (" ");
00713
00714
            return NULL;
00715
00716
        int ret = 0;
00717
        ret = pthread_mutex_init (&p->status_m, NULL);
00718
        if (ret)
00719
        {
            free (p);
LOG_END (" ");
00720
00721
00722
            return NULL:
00723
00724
        ret = pthread_cond_init (&p->status_c, NULL);
00725
        if (ret)
00726
         {
            free (p);
LOG_END ("_");
00727
00728
00729
            return NULL;
00730
00731
00732
            = (pthread_mutex_t *)malloc (n_threads * sizeof (pthread_mutex_t));
00733
        if (!p->m_rings)
00734
            free (p);
LOG_END (" ");
00735
00736
00737
            return NULL;
00738
        for (uint32_t i = 0; i < n_threads; ++i)</pre>
00739
00740
            pthread_mutex_init (&(p->m_rings[i]), NULL);
00741
00742
00743
00744
        n\_threads = (n\_threads < 2 ? 2 : n\_threads);
        n_threads = (n_threads > NUM_CORES ? NUM_CORES : n_threads);
00745
00746
        p->n_threads = n_threads;
00747
        p->status = 0;
00748
        p->list = NULL;
00749
        p->cb.on_connected = NULL;
00750
        p->cb.on_connected_arg = NULL;
00751
        p->cb.on_readed = NULL;
00752
        p->cb.on_readed_arg = NULL;
00753
        p->cb.on_wrote = NULL;
00754
        p->cb.on_wrote_arg = NULL;
00755
        p->cb.on_closed = NULL;
00756
        p->cb.on_closed_arg = NULL;
00757
        p->cb.on_error = NULL;
00758
        p->cb.on_error_arg = NULL;
        p->next = 0;
p->efds = (int *)malloc (sizeof (int) * p->n_threads);
00759
00760
00761
        if (!p->efds)
00762
00763
            free (p->m_rings);
00764
            free (p);
LOG_END (" ");
00765
00766
            return NULL;
00767
00768
        for (uint32_t i = 0; i < p->n_threads; ++i)
00769
            p->efds[i] = eventfd (0, EFD_NONBLOCK);
00770
00771
            if (p->efds[i] == ERROR_CODE)
00772
00773
                for (uint32_t j = 0; j < i; ++j)
00774
00775
                    close (p->efds[j]);
00776
00777
                free (p->efds);
00778
                free (p->m_rings);
```

7.14 low_saurion.c 91

```
free (p);
LOG_END (" ");
00780
00781
                 return NULL;
00782
00783
00784
        p->rings
00785
             = (struct io_uring *)malloc (sizeof (struct io_uring) * p->n_threads);
00786
        if (!p->rings)
00787
             for (uint32_t j = 0; j < p->n_threads; ++j)
00788
00789
00790
                close (p->efds[j]);
00791
00792
             free (p->efds);
00793
             free (p->m_rings);
            free (p);
LOG_END (" ");
00794
00795
00796
            return NULL;
00798
        for (uint32_t i = 0; i < p->n_threads; ++i)
00799
00800
             memset (&p->rings[i], 0, sizeof (struct io_uring));
             ret = io_uring_queue_init (SAURION_RING_SIZE, &p->rings[i], 0);
00801
00802
             if (ret)
00803
              {
                 for (uint32_t j = 0; j < p->n_threads; ++j)
00804
00805
00806
                     close (p->efds[j]);
00807
00808
                 free (p->efds);
00809
                 free (p->rings);
00810
                 free (p->m_rings);
                 free (p);
LOG_END (" ");
00811
00812
00813
                 return NULL;
00814
00815
00816
        p->pool = threadpool_create (p->n_threads);
00817
        LOG_END (" ");
00818 return p;
00819 }
00820
00821 [[nodiscard]]
00822 static int
00823 saurion_worker_master_loop_it (struct saurion \starconst s,
00824
                                        struct sockaddr_in *client_addr,
00825
                                        socklen_t *client_addr_len)
00826 {
        LOG_INIT (" ");
00827
        struct io_uring ring = s->rings[0];
struct io_uring_cqe *cqe = NULL;
00828
00830
        int ret = io_uring_wait_cqe (&ring, &cqe);
00831
        if (ret < 0)
00832
            free (cqe);
LOG_END (" ");
00833
00834
             return CRITICAL_CODE;
00836
00837
        struct request *req = (struct request *)cqe->user_data;
00838
        if (!req)
00839
         {
            io_uring_cqe_seen (&s->rings[0], cqe);
LOG_END (" ");
00840
00841
00842
            return SUCCESS_CODE;
00843
00844
        if (cqe->res < 0)</pre>
00845
            list_delete_node (&s->list, req);
LOG_END (" ");
00846
00847
            return CRITICAL_CODE;
00848
00849
00850
        if (req->client_socket == s->efds[0])
00851
00852
            io_uring_cqe_seen (&s->rings[0], cqe);
             list_delete_node (&s->list, req);
LOG_END (" ");
00853
00854
00855
            return ERROR_CODE;
00856
00857
        io_uring_cqe_seen (&s->rings[0], cqe);
00858
        switch (req->event_type)
00859
00860
          case EV_ACC:
00861
            handle_accept (s, cqe->res);
00862
             add_accept (s, client_addr, client_addr_len);
00863
             add_read (s, cqe->res);
00864
             list_delete_node (&s->list, req);
00865
            break:
```

```
case EV_REA:
00867
            if (cqe->res < 0)</pre>
00868
00869
                 handle_error (s, req);
00870
00871
             if (cge->res < 1)
00873
                 handle_close (s, req);
00874
00875
             if (cqe->res > 0)
00876
              {
00877
                handle read (s. reg);
00878
00879
             list_delete_node (&s->list, req);
            break;
00880
00881
          case EV_WRI:
            handle_write (s, req->client_socket);
list_delete_node (&s->list, req);
00882
00883
00884
             break;
00885
00886
        LOG_END (" ");
00887
        return SUCCESS_CODE;
00888 }
00889
00890 void
00891 saurion_worker_master (void *arg)
00892 {
00893
        LOG_INIT (" ");
        struct saurion *const s = (struct saurion <math>*)arg;
00894
        struct sockaddr_in client_addr;
00895
00896
        socklen t client addr len = sizeof (client addr);
00897
00898
        add_efd (s, s->efds[0], 0);
00899
        add_accept (s, &client_addr, &client_addr_len);
00900
        pthread_mutex_lock (&s->status_m);
00901
00902
        ++s->status;
        pthread_cond_broadcast (&s->status_c);
00904
        pthread_mutex_unlock (&s->status_m);
00905
        while (1)
00906
00907
            int ret
                 = saurion_worker_master_loop_it (s, &client_addr, &client_addr_len);
00908
             if (ret == ERROR_CODE || ret == CRITICAL_CODE)
00909
00910
              {
00911
                 break;
00912
               }
00913
00914
        pthread_mutex_lock (&s->status_m);
00915
        --s->status;
        pthread_cond_signal (&s->status_c);
00916
00917
        pthread_mutex_unlock (&s->status_m);
00918
        LOG_END (" ");
00919
        return;
00920 }
00921
00922 [[nodiscard]]
00923 static int
00924 saurion_worker_slave_loop_it (struct saurion *const s, const int sel)
00925 {
        LOG INIT (" "):
00926
        struct io_uring ring = s->rings[sel];
struct io_uring_cqe *cqe = NULL;
00927
00928
00929
00930
        add_efd (s, s->efds[sel], sel);
        int ret = io_uring_wait_cqe (&ring, &cqe);
if (ret < 0)</pre>
00931
00932
00933
          {
            free (cqe);
LOG_END (" ");
00934
00935
00936
            return CRITICAL_CODE;
00937
00938
        struct request *req = (struct request *)cqe->user_data;
00939
        if (!req)
00940
          {
00941
             io_uring_cqe_seen (&ring, cqe);
00942
             LOG_END (" ");
00943
            return SUCCESS_CODE;
00944
00945
        if (cqe->res < 0)</pre>
00946
             list_delete_node (&s->list, req);
LOG_END (" ");
00947
00948
00949
             return CRITICAL_CODE;
00950
        if (req->client_socket == s->efds[sel])
00951
00952
```

7.14 low_saurion.c 93

```
io_uring_cqe_seen (&ring, cqe);
            list_delete_node (&s->list, req);
LOG_END (" ");
00954
00955
            return ERROR_CODE;
00956
00957
00958
        io_uring_cqe_seen (&ring, cqe);
        switch (req->event_type)
00960
00961
          case EV_REA:
00962
            if (cqe->res < 0)</pre>
00963
              {
00964
                handle_error (s, req);
00965
00966
               (cqe->res < 1)
00967
              {
00968
                handle_close (s, req);
00969
00970
            if (cqe->res > 0)
00971
00972
                handle_read (s, req);
00973
00974
            list_delete_node (&s->list, req);
          break;
case EV_WRI:
handle_write (s, req->client_socket);
00975
00976
00977
00978
             list_delete_node (&s->list, req);
00979
00980
        LOG_END (" ");
00981
        return SUCCESS_CODE;
00982
00983 }
00984
00985 void
00986 saurion_worker_slave (void *arg)
00987 {
        LOG_INIT (" ");
00988
00989
        struct saurion_wrapper *const ss = (struct saurion_wrapper *)arg;
        struct saurion *s = ss->s;
00991
        const int sel = ss->sel;
00992
        free (ss);
00993
00994
        add_efd (s, s->efds[sel], sel);
00995
00996
        pthread_mutex_lock (&s->status_m);
00997
        ++s->status;
00998
        pthread_cond_broadcast (&s->status_c);
00999
        pthread_mutex_unlock (&s->status_m);
01000
        while (1)
01001
01002
             int res = saurion_worker_slave_loop_it (s, sel);
             if (res == ERROR_CODE || res == CRITICAL_CODE)
01003
01004
01005
                break;
01006
              }
01007
01008
        pthread_mutex_lock (&s->status_m);
01009
         --s->status;
01010
        pthread_cond_signal (&s->status_c);
01011
        pthread_mutex_unlock (&s->status_m);
01012
        LOG_END (" ");
01013
        return:
01014 }
01015
01016 [[nodiscard]]
01017 int
01018 saurion_start (struct saurion *const s)
01019 {
01020
        pthread_mutex_init (&print_mutex, NULL);
01021
        threadpool init (s->pool);
        threadpool_add (s->pool, saurion_worker_master, s);
struct saurion_wrapper *ss = NULL;
01023
01024
        for (uint32_t i = 1; i < s->n_threads; ++i)
01025
            ss = (struct saurion_wrapper *)malloc (sizeof (struct saurion_wrapper));
01026
            if (!ss)
01027
01028
01029
                return ERROR_CODE;
01030
            ss->s = s;
01031
             ss->sel = i;
01032
            threadpool_add (s->pool, saurion_worker_slave, ss);
01033
01034
01035
        pthread_mutex_lock (&s->status_m);
01036
        while (s->status < (int)s->n_threads)
01037
         {
01038
            pthread_cond_wait (&s->status_c, &s->status_m);
01039
```

```
pthread_mutex_unlock (&s->status_m);
01041
        return SUCCESS_CODE;
01042 }
01043
01044 void
01045 saurion stop (const struct saurion *const s)
01046 {
01047
        uint64_t u = 1;
01048
        for (uint32_t i = 0; i < s->n_threads; ++i)
01049
01050
            while (write (s->efds[i], &u, sizeof (u)) < 0)</pre>
01051
01052
                nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
01053
01054
01055
       threadpool_wait_empty (s->pool);
01056 }
01057
01058 void
01059 saurion_destroy (struct saurion *const s)
01060 {
01061
        pthread_mutex_lock (&s->status_m);
01062
        while (s->status > 0)
01063
01064
            pthread_cond_wait (&s->status_c, &s->status_m);
01065
       pthread_mutex_unlock (&s->status_m);
01066
01067
        threadpool_destroy (s->pool);
        for (uint32_t i = 0; i < s->n_threads; ++i)
01068
01069
01070
            io_uring_queue_exit (&s->rings[i]);
01071
           pthread_mutex_destroy (&s->m_rings[i]);
01072
01073
       free (s->m_rings);
        list_free (&s->list);
for (uint32_t i = 0; i < s->n_threads; ++i)
01074
01075
01076
            close (s->efds[i]);
01078
01079
       free (s->efds);
01080
        if (!s->ss)
        {
01081
01082
            close (s->ss);
01083
       free (s->rings);
01085
       pthread_mutex_destroy (&s->status_m);
01086 pthread_cond_destroy (&s->status_c);
01087
       free (s);
01088 }
01089
01090 void
01091 saurion_send (struct saurion *const s, const int fd, const char *const msg)
01092 {
01093
        add_write (s, fd, msg, next (s));
01094 }
```

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
            counter++;
            if (i % 10000 == 0)
00013
00014
               printf ("Thread %d at iteration %d\n", id, i);
00015
00016
00017
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
00029
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
00046
            fprintf (stderr, "Error joining thread 1\n");
            return 2;
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
       printf ("Thread 2 joined\n");
00057
00058
        printf ("Final counter value: %d\n", counter);
00059
00060
        return 0;
00061 }
```

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 {
00007
        this->s = saurion_create (thds);
80000
       if (!this->s)
00009
00010
            return:
00011
00012
       this->s->ss = sck;
```

```
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
00028 {
00029
        saurion_stop (this->s);
00030 }
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;
00052 s->cb.on_wrote_arg = arg;
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
        saurion_send (this->s, fd, msg);
```

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 <stdib.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.
        LOG_INIT (" ");
00109
        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
                pthread_cond_wait (&pool->queue_cond, &pool->queue_lock);
00116
00117
00118
00119
            if (pool->stop && pool->task_queue_head == NULL)
00120
00121
                pthread_mutex_unlock (&pool->queue_lock);
00122
                break;
00123
00124
00125
            struct task *task = pool->task_queue_head;
00126
            if (task != NULL)
00127
00128
                pool->task_queue_head = task->next;
00129
                if (pool->task_queue_head == NULL)
00130
                  pool->task gueue 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);
00143
00144
        LOG_END (" ");
pthread_exit (NULL);
00145
00146
00147
        return NULL;
00148 }
```

7.20 threadpool.c

Go to the documentation of this file.

```
00001 #include "threadpool.h"
00002 #include "config.h"
                                      // for NUM_CORES
00002 #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 *);
00014
          void *argument;
00015
          struct task *next;
00016 };
00017
00018 struct threadpool
00019 {
         pthread_t *threads;
00021
          size_t num_threads;
00022
          struct task *task_queue_head;
00023
          struct task *task_queue_tail;
00024
          pthread_mutex_t queue_lock;
00025
         pthread_cond_t queue_cond;
00026
         pthread_cond_t empty_cond;
00027
          int stop;
```

7.20 threadpool.c 99

```
00028
        int started;
00029 };
00030
00031 struct threadpool \star
00032 threadpool_create (size_t num_threads)
00033 {
        LOG_INIT (" ");
00035
        struct threadpool *pool = malloc (sizeof (struct threadpool));
        if (pool == NULL)
00036
00037
00038
             perror ("Failed to allocate threadpool");
             LOG_END (" ");
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
        pool->num_threads = num_threads;
pool->threads = malloc (sizeof (pthread_t) * num_threads);
00051
00052
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;
        pool->stop = FALSE;
pool->started = FALSE;
00063
00064
00065
00066
         if (pthread_mutex_init (&pool->queue_lock, NULL) != 0)
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);
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
        LOG_END (" ");
00096
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 (" ");
00110
        struct threadpool *pool = (struct threadpool *)arg;
00111
        while (TRUE)
00112
             pthread_mutex_lock (&pool->queue_lock);
00113
00114
             while (pool->task queue head == NULL && !pool->stop)
```

```
{
00116
                pthread_cond_wait (&pool->queue_cond, &pool->queue_lock);
00117
00118
00119
            if (pool->stop && pool->task_queue_head == NULL)
00120
              {
00121
                pthread_mutex_unlock (&pool->queue_lock);
00122
00123
00124
00125
            struct task *task = pool->task_queue_head;
            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)
00134
                    pthread_cond_signal (&pool->empty_cond);
00135
00136
00137
            pthread_mutex_unlock (&pool->queue_lock);
00138
00139
            if (task != NULL)
00140
00141
                task->function (task->argument);
00142
                free (task);
00143
00144
00145
       LOG_END (" ");
00146
       pthread_exit (NULL);
00147
        return NULL;
00148 }
00149
00150 void
00151 threadpool init (struct threadpool *pool)
00153
        LOG_INIT (" ");
00154
        if (pool == NULL || pool->started)
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
                != 0)
00164
                perror ("Failed to create thread");
00165
00166
                pool->stop = TRUE;
00167
                break;
00168
             }
00169
       pool->started = TRUE;
LOG_END (" ");
00170
00171
00172 }
00173
00174 void
00175 threadpool_add (struct threadpool *pool, void (*function) (void *),
00176
                      void *argument)
00177 {
00178
       LOG_INIT (" ");
00179
        if (pool == NULL || function == NULL)
00180
           LOG_END (" ");
00181
00182
           return:
00183
00184
00185
        struct task *new_task = malloc (sizeof (struct task));
00186
        if (new_task == NULL)
00187
         {
           perror ("Failed to allocate task");
LOG_END (" ");
00188
00189
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
00201
            pool->task gueue head = new task;
```

7.20 threadpool.c 101

```
pool->task_queue_tail = new_task;
00203
00204
        else
00205
        {
           pool->task_queue_tail->next = new_task;
00206
           pool->task_queue_tail = new_task;
00207
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 *pool)
00217 {
        LOG_INIT (" ");
00218
        if (pool == NULL || !pool->started)
00219
00221
            LOG_END (" ");
00222
           return;
00223
00224
       threadpool_wait_empty (pool);
00225
00226
       pthread_mutex_lock (&pool->queue_lock);
00227
        pool->stop = TRUE;
00228
        pthread_cond_broadcast (&pool->queue_cond);
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
00235
        pool->started = FALSE;
00236
        LOG_END (" ");
00237 }
00238
00239 int
00240 threadpool_empty (struct threadpool *pool)
00241 {
        LOG_INIT (" ");
00242
        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);
        LOG_END (" ");
00251
00252
       return empty;
00253 }
00254
00255 void
00256 threadpool_wait_empty (struct threadpool *pool)
00257 {
00258
       LOG_INIT (" ");
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
            pthread_cond_wait (&pool->empty_cond, &pool->queue_lock);
00268
00269
        pthread_mutex_unlock (&pool->queue_lock);
        LOG_END (" ");
00270
00271 }
00272
00273 void
00274 threadpool_destroy (struct threadpool *pool)
00275 {
00276
        LOG_INIT (" ");
00277
        if (pool == NULL)
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;
        while (task != NULL)
00286
00287
00288
            struct task *tmp = task;
```

Index

/_w/saurion/saurion/include/linked_list.h, 53, 55	Saurion, 38
/_w/saurion/saurion/include/low_saurion.h, 55, 62	EV_ACC
/_w/saurion/saurion/include/low_saurion_secret.h, 62,	low_saurion.c, 72
63	EV_ERR
/_w/saurion/saurion/include/saurion.hpp, 63, 64	low_saurion.c, 72
/_w/saurion/saurion/include/threadpool.h, 64, 65	EV_REA
/_w/saurion/saurion/src/linked_list.c, 65, 68	low_saurion.c, 72
/_w/saurion/saurion/src/low_saurion.c, 70, 82	EV_WAI
/_w/saurion/saurion/src/main.c, 94	low_saurion.c, 72
/_w/saurion/saurion/src/saurion.cpp, 95	EV_WRI
/_w/saurion/saurion/src/threadpool.c, 96, 98	low_saurion.c, 72
POSIX_C_SOURCE	event_type
LowSaurion, 12	request, 32
~Saurion	•
Saurion, 39	FALSE
,	threadpool.c, 97
add_accept	free_node
low_saurion.c, 73	linked_list.c, 66
add_efd	free_request
low_saurion.c, 73	LowSaurion, 13
add fd	function
low_saurion.c, 74	task, 50
add_read	,
low_saurion.c, 74	handle_accept
add_read_continue	low_saurion.c, 76
low_saurion.c, 74	handle_close
add_write	low_saurion.c, 76
low_saurion.c, 75	handle_error
allocate_iovec	low_saurion.c, 76
LowSaurion, 12	handle_read
argument	low_saurion.c, 76
task, 50	handle_write
	low saurion.c, 77
children	htonll
Node, 31	low_saurion.c, 77
client_socket	-
request, 32	init
ClosedCb	Saurion, 40
Saurion, 38	initialize_iovec
ConnectedCb	LowSaurion, 13
Saurion, 38	iov
create_node	request, 33
linked_list.c, 66	iovec_count
30 <u>_</u> , 30	request, 33
efds	•
low_saurion.h, 56	linked_list.c
saurion, 35	create_node, 66
empty_cond	free_node, 66
threadpool, 51	list_delete_node, 67
ErrorCb	list_free, 67

list_insert, 67	on_error, 58
list_mutex, 68	on_error_arg, 59
linked_list.h	on_readed, 59
list_delete_node, 53	on_readed_arg, 59
list_free, 54	on_wrote, 60
list_insert, 54	on_wrote_arg, 60
list	pool, 60
low_saurion.h, 57	rings, 60
saurion, 35	ss, 61
list delete node	status, 61
linked list.c, 67	status c, 61
linked_list.h, 53	status m, 61
list free	LowSaurion, 9
linked list.c, 67	_POSIX_C_SOURCE, 12
linked list.h, 54	allocate iovec, 12
list insert	free_request, 13
linked list.c, 67	initialize iovec, 13
linked list.h, 54	PACKING SZ, 12
list mutex	read_chunk, 15
linked list.c, 68	saurion create, 18
low saurion.c	saurion destroy, 20
-	saurion send, 20
add_accept, 73	-
add_efd, 73	saurion_set_socket, 22
add_fd, 74	saurion_start, 22
add_read, 74	saurion_stop, 23
add_read_continue, 74	set_request, 24
add_write, 75	m rings
EV_ACC, 72	_ •
EV_ERR, 72	low_saurion.h, 57
EV_REA, 72	saurion, 35 MAX
EV_WAI, 72	
EV_WRI, 72	low_saurion.c, 72
handle_accept, 76	MIN
handle_close, 76	low_saurion.c, 73
handle_error, 76	n threads
handle_read, 76	low saurion.h, 57
handle_write, 77	-
htonll, 77	saurion, 35
MAX, 72	next
MIN, 73	low_saurion.c, 78
next, 78	low_saurion.h, 57
ntohll, 78	Node, 31
print_mutex, 81	saurion, 35
saurion_worker_master, 78	task, 50
saurion_worker_master_loop_it, 79	next_iov
saurion_worker_slave, 80	request, 33
saurion_worker_slave_loop_it, 80	next_offset
TIMEOUT RETRY SPEC, 81	request, 33
low saurion.h	Node, 31
efds, 56	children, 31
list, 57	next, 31
m_rings, 57	ptr, 31
n_threads, 57	size, 32
next, 57	ntohll
on closed, 57	low_saurion.c, 78
on_closed_arg, 58	num_threads
on connected, 58	threadpool, 51
on_connected_arg, 58	
on_oonnoolou_arg, oo	on_closed

low_saurion.h, 57		low_saurion.c, 81
Saurion, 40	ptr	N 1 04
saurion::saurion_callbacks, 43		Node, 31
saurion_callbacks, 46	anei	ie cond
on_closed_arg	quec	threadpool, 52
low_saurion.h, 58	anei	ie lock
saurion::saurion_callbacks, 43	quec	threadpool, 52
saurion_callbacks, 47 on connected		in oddpool, oz
low_saurion.h, 58	read	_chunk
Saurion, 40		LowSaurion, 15
saurion::saurion_callbacks, 43	Read	dedCb
saurion_callbacks, 47		Saurion, 38
on_connected_arg	requ	est, 32
low_saurion.h, 58		client_socket, 32
saurion::saurion_callbacks, 44		event_type, 32
saurion_callbacks, 47		iov, 33
on error		iovec_count, 33
low_saurion.h, 58		next_iov, 33
Saurion, 40		next_offset, 33
saurion::saurion_callbacks, 44		prev, 33
saurion_callbacks, 47		prev_remain, 33
on_error_arg		prev_size, 34
low_saurion.h, 59	rings	3
saurion::saurion_callbacks, 44		low_saurion.h, 60
saurion_callbacks, 48		saurion, 36
on readed		
low_saurion.h, 59	S	
Saurion, 40		Saurion, 42
saurion::saurion_callbacks, 44	_	saurion_wrapper, 49
saurion_callbacks, 48	Saur	ion, 37
on_readed_arg		~Saurion, 39
low_saurion.h, 59		ClosedCb, 38
saurion::saurion_callbacks, 45		ConnectedCb, 38
saurion_callbacks, 48		ErrorCb, 38
on_wrote		init, 40
low_saurion.h, 60		on_closed, 40
Saurion, 41		on_connected, 40
saurion::saurion_callbacks, 45		on_error, 40
saurion_callbacks, 48		on_readed, 40 on wrote, 41
on_wrote_arg		operator=, 41
low_saurion.h, 60		ReadedCb, 38
saurion::saurion_callbacks, 45		s, 42
saurion_callbacks, 49		Saurion, 39
operator=		send, 41
Saurion, 41		stop, 41
DAOI/INO 07		WroteCb, 38
PACKING_SZ	saur	ion, 34
LowSaurion, 12	Saur	efds, 35
pool		list, 35
low_saurion.h, 60		m_rings, 35
saurion, 36		n_threads, 35
prev request, 33		next, 35
prev_remain		pool, 36
request, 33		rings, 36
prev_size		ss, 36
request, 34		status, 36
print_mutex		status_c, 36
Print_motox		<i>- '</i>

status_m, 37	threadpool, 52
saurion::saurion_callbacks, 42	status
on_closed, 43	low_saurion.h, 61
on_closed_arg, 43	saurion, 36
on_connected, 43	status_c
on_connected_arg, 44	low_saurion.h, 61
on_error, 44	saurion, <mark>36</mark>
on_error_arg, 44	status_m
on readed, 44	low saurion.h, 61
on readed arg, 45	saurion, 37
on wrote, 45	stop
on_wrote_arg, 45	Saurion, 41
saurion_callbacks, 46	threadpool, 52
on closed, 46	
on_closed_arg, 47	task, 50
on connected, 47	argument, 50
on_connected_arg, 47	function, 50
on error, 47	next, 50
on error arg, 48	task queue head
on readed, 48	threadpool, 52
-	task_queue_tail
on_readed_arg, 48	threadpool, 52
on_wrote, 48	ThreadPool, 25
on_wrote_arg, 49	threadpool_add, 26
saurion_create	threadpool create, 26
LowSaurion, 18	threadpool_create_default, 27
saurion_destroy	threadpool_destroy, 27
LowSaurion, 20	threadpool_empty, 28
saurion_send	. –
LowSaurion, 20	threadpool_init, 28
saurion_set_socket	threadpool_stop, 29
LowSaurion, 22	threadpool_wait_empty, 29
saurion_start	threadpool, 51
LowSaurion, 22	empty_cond, 51
saurion_stop	num_threads, 51
LowSaurion, 23	queue_cond, 52
saurion_worker_master	queue_lock, 52
low_saurion.c, 78	started, 52
saurion_worker_master_loop_it	stop, 52
low_saurion.c, 79	task_queue_head, 52
saurion_worker_slave	task_queue_tail, 52
low_saurion.c, 80	threads, 52
saurion_worker_slave_loop_it	threadpool.c
low_saurion.c, 80	FALSE, 97
saurion_wrapper, 49	threadpool_worker, 97
s, 49	TRUE, 97
sel, 49	threadpool_add
sel	ThreadPool, 26
saurion_wrapper, 49	threadpool_create
send	ThreadPool, 26
Saurion, 41	threadpool_create_default
set_request	ThreadPool, 27
LowSaurion, 24	threadpool_destroy
size	ThreadPool, 27
	threadpool_empty
Node, 32	. — . ,
cc	ThreadPool, 28
SS low couries b. 61	ThreadPool, 28 threadpool init
low_saurion.h, 61	threadpool_init
low_saurion.h, 61 saurion, 36	threadpool_init ThreadPool, 28
low_saurion.h, 61	threadpool_init

threadpool_wait_empty
 ThreadPool, 29
threadpool_worker
 threadpool.c, 97
threads
 threadpool, 52
TIMEOUT_RETRY_SPEC
 low_saurion.c, 81
TRUE
 threadpool.c, 97
WroteCb
 Saurion, 38