### 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
4.1 File List	. 7
5 Module Documentation	9
5.1 LowSaurion	. 9
5.1.1 Detailed Description	. 10
5.1.2 Macro Definition Documentation	
5.1.2.1 PACKING_SZ	. 12
5.1.3 Function Documentation	
5.1.3.1 allocate_iovec()	. 12
5.1.3.2 EXTERNAL_set_socket()	
5.1.3.3 free_request()	
5.1.3.4 initialize_iovec()	
5.1.3.5 read_chunk()	
5.1.3.6 saurion_create()	
5.1.3.7 saurion_destroy()	
5.1.3.8 saurion_send()	
5.1.3.9 saurion_start()	
5.1.3.10 saurion_stop()	
5.1.3.11 set_request()	
6 Class Documentation	25
6.1 ThreadPool::AsyncMultiQueue Struct Reference	
6.1.1 Detailed Description	
6.1.2 Constructor & Destructor Documentation	
6.1.2.1 AsyncMultiQueue() [1/3]	
6.1.2.2 ~AsyncMultiQueue()	
6.1.2.3 AsyncMultiQueue() [2/3]	
6.1.2.4 AsyncMultiQueue() [3/3]	
6.1.3 Member Function Documentation	
6.1.3.1 clear()	
6.1.3.2 empty()	
6.1.3.3 front()	
6.1.3.5 operator=() [1/2]	
6.1.3.6 operator=() [2/2]	. 28

6.1.3.7 pop()	28
6.1.3.8 push()	28
6.1.3.9 remove_queue()	28
6.1.4 Member Data Documentation	28
6.1.4.1 m_it	29
6.1.4.2 m_queues	29
6.2 ThreadPool::AsyncQueue Struct Reference	29
6.2.1 Detailed Description	29
6.2.2 Constructor & Destructor Documentation	29
<b>6.2.2.1 AsyncQueue()</b> [1/3]	30
6.2.2.2 ~AsyncQueue()	30
<b>6.2.2.3 AsyncQueue()</b> [2/3]	30
<b>6.2.2.4 AsyncQueue()</b> [3/3]	30
6.2.3 Member Function Documentation	30
6.2.3.1 empty()	30
6.2.3.2 front()	31
<b>6.2.3.3</b> operator=() [1/2]	31
6.2.3.4 operator=() [2/2]	31
6.2.3.5 pop()	31
6.2.3.6 push()	31
6.2.4 Member Data Documentation	32
6.2.4.1 m_cnt	32
6.2.4.2 m_max	32
6.2.4.3 m_mtx	32
6.2.4.4 m_queue	32
6.3 Node Struct Reference	32
6.3.1 Detailed Description	33
6.3.2 Member Data Documentation	33
6.3.2.1 children	33
6.3.2.2 next	33
6.3.2.3 ptr	33
6.3.2.4 size	33
6.4 request Struct Reference	34
6.4.1 Detailed Description	34
6.4.2 Member Data Documentation	34
6.4.2.1 client_socket	34
6.4.2.2 event_type	34
6.4.2.3 iov	34
6.4.2.4 iovec_count	35
6.4.2.5 next_iov	35
6.4.2.6 next_offset	35
6.4.2.7 prev	35

6.4.2.8 prev_remain	35
6.4.2.9 prev_size	35
6.5 saurion Struct Reference	36
6.5.1 Detailed Description	36
6.5.2 Member Data Documentation	36
6.5.2.1 efds	36
6.5.2.2 list	37
6.5.2.3 m_rings	37
6.5.2.4 n_threads	37
6.5.2.5 next	37
6.5.2.6 pool	37
6.5.2.7 rings	38
6.5.2.8 ss	38
6.5.2.9 status	38
6.5.2.10 status_c	38
6.5.2.11 status_m	38
6.6 Saurion Class Reference	39
6.6.1 Detailed Description	
6.6.2 Member Typedef Documentation	
6.6.2.1 ClosedCb	39
6.6.2.2 ConnectedCb	40
6.6.2.3 ErrorCb	40
6.6.2.4 ReadedCb	
6.6.2.5 WroteCb	40
6.6.3 Constructor & Destructor Documentation	40
<b>6.6.3.1 Saurion()</b> [1/3]	
6.6.3.2 ∼Saurion()	41
<b>6.6.3.3 Saurion()</b> [2/3]	41
<b>6.6.3.4 Saurion()</b> [3/3]	41
6.6.4 Member Function Documentation	41
6.6.4.1 init()	
6.6.4.2 on_closed()	
6.6.4.3 on_connected()	
6.6.4.4 on_error()	
6.6.4.5 on_readed()	
6.6.4.6 on_wrote()	
6.6.4.7 operator=() [1/2]	
6.6.4.8 operator=() [2/2]	
6.6.4.9 send()	
6.6.4.10 stop()	
6.6.5 Member Data Documentation	
6.6.5.1 s	43

6.7 saurion::saurion_callbacks Struct Reference	44
6.7.1 Detailed Description	44
6.7.2 Member Data Documentation	44
6.7.2.1 on_closed	44
6.7.2.2 on_closed_arg	45
6.7.2.3 on_connected	45
6.7.2.4 on_connected_arg	45
6.7.2.5 on_error	45
6.7.2.6 on_error_arg	46
6.7.2.7 on_readed	46
6.7.2.8 on_readed_arg	46
6.7.2.9 on_wrote	46
6.7.2.10 on_wrote_arg	47
6.8 saurion_callbacks Struct Reference	47
6.8.1 Detailed Description	48
6.8.2 Member Data Documentation	48
6.8.2.1 on_closed	48
6.8.2.2 on_closed_arg	48
6.8.2.3 on_connected	48
6.8.2.4 on_connected_arg	49
6.8.2.5 on_error	49
6.8.2.6 on_error_arg	49
6.8.2.7 on_readed	49
6.8.2.8 on_readed_arg	50
6.8.2.9 on_wrote	50
6.8.2.10 on_wrote_arg	50
6.9 saurion_wrapper Struct Reference	51
6.9.1 Detailed Description	51
6.9.2 Member Data Documentation	51
6.9.2.1 s	51
6.9.2.2 sel	51
6.10 Task Struct Reference	51
6.10.1 Detailed Description	52
6.10.2 Constructor & Destructor Documentation	52
6.10.2.1 Task() [1/3]	52
6.10.2.2 Task() [2/3]	52
<b>6.10.2.3 Task()</b> [3/3]	52
6.10.2.4 ∼Task()	52
6.10.3 Member Function Documentation	52
6.10.3.1 operator=() [1/2]	53
6.10.3.2 operator=() [2/2]	53
6.10.4 Member Data Documentation	53

6.10.4.1 argument	53
6.10.4.2 function	53
6.11 ThreadPool Class Reference	53
6.11.1 Detailed Description	55
6.11.2 Member Typedef Documentation	55
6.11.2.1 AsyncMultiQueue	55
6.11.2.2 AsyncQueue	55
6.11.3 Constructor & Destructor Documentation	55
6.11.3.1 ThreadPool() [1/4]	55
6.11.3.2 ThreadPool() [2/4]	55
<b>6.11.3.3 ThreadPool()</b> [3/4]	56
6.11.3.4 ThreadPool() [4/4]	56
6.11.3.5 ~ThreadPool()	56
6.11.4 Member Function Documentation	56
6.11.4.1 add() [1/2]	56
<b>6.11.4.2 add()</b> [2/2]	57
6.11.4.3 empty()	57
6.11.4.4 init()	57
6.11.4.5 new_queue()	57
6.11.4.6 operator=() [1/2]	58
6.11.4.7 operator=() [2/2]	58
6.11.4.8 remove_queue()	58
6.11.4.9 stop()	58
6.11.4.10 thread_entry()	59
6.11.4.11 thread_worker()	59
6.11.4.12 wait_closeable()	59
6.11.4.13 wait_empty()	60
6.11.5 Member Data Documentation	60
6.11.5.1 m_faccept	60
6.11.5.2 m_fstop	60
6.11.5.3 m_nth	60
6.11.5.4 m_q_cond	60
6.11.5.5 m_q_mtx	61
6.11.5.6 m_queues	61
6.11.5.7 m_started	61
6.11.5.8 m_ths	61
6.11.5.9 s_mtx	61
7 File Documentation	63
7.1 /_w/saurion/saurion/include/cthreadpool.hpp File Reference	63
7.1.1 Typedef Documentation	63
7.1.1.1 ThreadPool	63

7.1.2 Function Documentation	,4
7.1.2.1 ThreadPool_add()	,4
7.1.2.2 ThreadPool_add_default()	,4
7.1.2.3 ThreadPool_create()	,4
7.1.2.4 ThreadPool_create_default()	,4
7.1.2.5 ThreadPool_destroy()	Ę
7.1.2.6 ThreadPool_empty()	5
7.1.2.7 ThreadPool_init()	Ę
7.1.2.8 ThreadPool_new_queue()	Ę
7.1.2.9 ThreadPool_remove_queue()	Ę
7.1.2.10 ThreadPool_stop()	6
7.1.2.11 ThreadPool_wait_empty()	6
7.2 cthreadpool.hpp	6
7.3 /w/saurion/saurion/include/linked_list.h File Reference	$\epsilon$
7.4 linked_list.h	7
7.5 /w/saurion/saurion/include/low_saurion.h File Reference	7
7.5.1 Variable Documentation	ξ
7.5.1.1 efds	ξ
7.5.1.2 list	ξ
7.5.1.3 m_rings	ć
7.5.1.4 n_threads	Ç
7.5.1.5 next	ć
7.5.1.6 on_closed	ć
7.5.1.7 on_closed_arg	ć
7.5.1.8 on_connected	(
7.5.1.9 on_connected_arg	(
7.5.1.10 on_error	(
7.5.1.11 on_error_arg	1
7.5.1.12 on_readed	1
7.5.1.13 on_readed_arg	1
7.5.1.14 on_wrote	1
7.5.1.15 on_wrote_arg	2
7.5.1.16 pool	2
7.5.1.17 rings	2
7.5.1.18 ss	2
7.5.1.19 status	3
7.5.1.20 status_c	3
7.5.1.21 status_m	:
7.6 low_saurion.h	:
7.7 /w/saurion/saurion/include/low_saurion_secret.h File Reference	2
7.8 low_saurion_secret.h	7
7.9 / w/saurion/saurion/include/saurion.hop File Beference	F

7.10 saurion.hpp
7.11 /w/saurion/saurion/include/threadpool.hpp File Reference
7.11.1 Enumeration Type Documentation
7.11.1.1 StopFlags
7.12 threadpool.hpp
7.13 /w/saurion/saurion/src/cthreadpool.cpp File Reference
7.13.1 Function Documentation
7.13.1.1 ThreadPool_add()
7.13.1.2 ThreadPool_add_default()
7.13.1.3 ThreadPool_create()
7.13.1.4 ThreadPool_create_default()
7.13.1.5 ThreadPool_destroy()
7.13.1.6 ThreadPool_empty()
7.13.1.7 ThreadPool_init()
7.13.1.8 ThreadPool_new_queue()
7.13.1.9 ThreadPool_remove_queue()
7.13.1.10 ThreadPool_stop()
7.13.1.11 ThreadPool_wait_empty()
7.14 cthreadpool.cpp
7.15 /w/saurion/src/linked_list.c File Reference
7.15.1 Function Documentation
7.15.1.1 create_node()
7.15.1.2 free_node()
7.15.1.3 list_delete_node()
7.15.1.4 list_free()
7.15.1.5 list_insert()
7.15.2 Variable Documentation
7.15.2.1 list_mutex
7.16 linked_list.c
7.17 /w/saurion/src/low_saurion.c File Reference
7.17.1 Macro Definition Documentation
7.17.1.1 EVENT_TYPE_ACCEPT
7.17.1.2 EVENT_TYPE_ERROR
7.17.1.3 EVENT_TYPE_READ
7.17.1.4 EVENT_TYPE_WAIT
7.17.1.5 EVENT_TYPE_WRITE
7.17.1.6 MAX
7.17.1.7 MIN
7.17.2 Function Documentation
7.17.2.1 add_accept()
7.17.2.2 add_efd()
7.17.2.3 add_read()

7.17.2.4 add_read_continue()	90
7.17.2.5 add_write()	91
7.17.2.6 handle_accept()	91
7.17.2.7 handle_close()	92
7.17.2.8 handle_error()	92
7.17.2.9 handle_read()	92
7.17.2.10 handle_write()	93
7.17.2.11 htonll()	93
7.17.2.12 next()	93
7.17.2.13 ntohll()	93
7.17.2.14 saurion_worker_master()	94
7.17.2.15 saurion_worker_slave()	94
7.17.3 Variable Documentation	95
7.17.3.1 print_mutex	95
7.18 low_saurion.c	96
7.19 /w/saurion/saurion/src/saurion.cpp File Reference	106
7.20 saurion.cpp	106
7.21 /w/saurion/saurion/src/threadpool.cpp File Reference	106
7.21.1 Typedef Documentation	107
7.21.1.1 T	107
7.21.1.2 TP	107
7.22 threadpool.cpp	107
Index	111

## **Todo List**

```
Member EXTERNAL_set_socket (int p)
Eliminar

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

# **Module Index**

### 2.1 Modules

Here is a list of all modules:	
LowSaurion	ç

4 Module Index

# **Class Index**

### 3.1 Class List

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

ThreadPool::AsyncMultiQueue	25
ThreadPool::AsyncQueue	29
Node	32
request	34
saurion	
Main structure for managing io_uring and socket events	36
Saurion	39
saurion::saurion_callbacks	
Structure containing callback functions to handle socket events	44
saurion_callbacks	
Structure containing callback functions to handle socket events	47
saurion_wrapper	
Task	51
ThreadPool	53

6 Class Index

# File Index

### 4.1 File List

Here is a list of all files with brief descriptions:

/w/saurion/saurion/include/cthreadpool.hpp	63
/w/saurion/saurion/include/linked_list.h	66
/w/saurion/saurion/include/low_saurion.h	67
/w/saurion/saurion/include/low_saurion_secret.h	74
/w/saurion/saurion/include/saurion.hpp	75
/w/saurion/saurion/include/threadpool.hpp	76
/w/saurion/src/cthreadpool.cpp	78
/w/saurion/src/linked_list.c	81
/w/saurion/src/low_saurion.c	86
/w/saurion/src/saurion.cpp	106
/ w/saurion/src/threadpool.cpp	106

8 File Index

### **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 PACKING SZ 128

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

#### **Functions**

- int EXTERNAL\_set\_socket (int p)
- struct saurion \* saurion\_create (uint32\_t n\_threads)

Creates an instance of the saurion structure.

• int saurion\_start (struct saurion \*s)

Starts event processing in the saurion structure.

void saurion\_stop (const struct saurion \*s)

Stops event processing in the saurion structure.

void saurion\_destroy (struct saurion \*s)

Destroys the saurion structure and frees all associated resources.

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

Sends a message through a socket using io\_uring.

- int allocate\_iovec (struct iovec \*iov, size\_t amount, size\_t pos, size\_t size, void \*\*chd\_ptr)
- int initialize\_iovec (struct iovec \*iov, size\_t amount, size\_t pos, const void \*msg, size\_t size, uint8\_t h)

Initializes a specified iovec structure with a message fragment.

• int set\_request (struct request \*\*r, struct Node \*\*I, size\_t s, const void \*m, uint8\_t h)

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

int read\_chunk (void \*\*dest, size\_t \*len, struct request \*const req)

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

void free\_request (struct request \*req, void \*\*children\_ptr, size\_t amount)

#### 5.1.1 Detailed Description

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

This function allocates memory for each struct iovec

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

The messages are composed of three main parts:

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

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

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

- iov\_base, which is an array where the chunk of the message is stored.
- iov\_len, the number of bytes used in the iov\_base array.

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

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

The structure of the message is as follows:

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

The structure of the iovec division is:

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

#### Basic usage example:

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

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

#### Author

Israel

#### Date

2024

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

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

#### **Parameters**

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

#### Return values

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

Note

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

#### 5.1.2 Macro Definition Documentation

#### 5.1.2.1 PACKING\_SZ

```
#define PACKING_SZ 128
```

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 128 bytes, meaning that structures marked with  $\_$ attribute $\_$ ((aligned ( $\leftarrow$  PACKING\_SZ))) will be aligned to 128-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 129 of file low saurion.h.

#### 5.1.3 Function Documentation

#### 5.1.3.1 allocate\_iovec()

#### Definition at line 114 of file low\_saurion.c.

```
00114
00115
        if (!iov || !chd_ptr) {
00116
          return ERROR_CODE;
00117
       iov->iov_base = malloc(CHUNK_SZ);
00118
00119
       if (!iov->iov base) {
00120
         return ERROR_CODE;
00121
       iov->iov_len = (pos == (amount - 1) ? (size % CHUNK_SZ) : CHUNK_SZ);
if (iov->iov_len == 0) {
00122
00123
         iov->iov_len = CHUNK_SZ;
00124
00125
        chd_ptr[pos] = iov->iov_base;
00126
00127
        return SUCCESS_CODE;
00128 }
```

#### 5.1.3.2 EXTERNAL\_set\_socket()

```
int EXTERNAL_set_socket ( \label{eq:condition} \text{int } p \ )
```

#### **Todo** Eliminar

```
Definition at line 548 of file low_saurion.c.
00548
00549
        int sock = 0;
00550
        struct sockaddr_in srv_addr;
00551
00552
        sock = socket(PF_INET, SOCK_STREAM, 0);
00553
        if (sock < 1) {
          return ERROR_CODE;
00554
00555
00556
00557
00558
        if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(int)) < 0) {</pre>
00559
          return ERROR_CODE;
00560
00561
        if (setsockopt(sock, SOL_SOCKET, SO_REUSEPORT, &enable, sizeof(int)) < 0) {</pre>
00562
          return ERROR_CODE;
00563
00564
00565
        memset(&srv_addr, 0, sizeof(srv_addr));
        srv_addr.sin_family = AF_INET;
srv_addr.sin_port = htons(p);
00566
00567
00568
        srv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
00569
00570
        /\star We bind to a port and turn this socket into a listening
00571 * socket.
00572 * */
00573
        if (bind(sock, (const struct sockaddr *)&srv_addr, sizeof(srv_addr)) < 0) {</pre>
00574
         return ERROR_CODE;
00575
00576
00577
       return ERROR_CODE;
}
        if (listen(sock, ACCEPT_QUEUE) < 0) {</pre>
00578
00579
00580
00581
        return sock;
00582 }
```

#### 5.1.3.3 free\_request()

```
void free_request (
               struct request * req,
               void ** children_ptr,
               size_t amount )
Definition at line 61 of file low_saurion.c.
00061
00062
        if (children_ptr) {
00063
          free(children_ptr);
00064
          children_ptr = NULL;
00065
00066
       for (size_t i = 0; i < amount; ++i) {</pre>
00067
         free(req->iov[i].iov_base);
00068
         req->iov[i].iov_base = NULL;
00069
00070
       free (reg):
00071
        req = NULL;
00072
        free(children_ptr);
00073
        children_ptr = NULL;
00074 }
```

#### 5.1.3.4 initialize\_iovec()

```
int initialize_iovec (
    struct iovec * iov,
    size_t amount,
    size_t pos,
    const void * msg,
    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 77 of file low\_saurion.c.

```
char *dest = (char *)iov->iov_base;
char *orig = (char *)msg + pos * CHUNK_SZ;
00085
00086
          size_t cpy_sz = 0;
00087
          if (h) {
            if (pos == 0) {
00088
00089
              uint64_t send_size = htonll(size);
              memcpy(dest, &send_size, sizeof(uint64_t));
00090
00091
              dest += sizeof(uint64_t);
00092
              len -= sizeof(uint64_t);
00093
            } else {
              orig -= sizeof(uint64_t);
00094
00095
00096
             if ((pos + 1) == amount) {
00097
00098
              cpy_sz = (len < size ? len : size);</pre>
00099
              dest[cpy_sz] = 0;
00100
00101
00102
          cpy_sz = (len < size ? len : size);
00103
          memcpy(dest, orig, cpy_sz);
00104
          dest += cpy_sz;
00105
          size_t rem = CHUNK_SZ - (dest - (char *)iov->iov_base);
00106
          memset(dest, 0, rem);
00107
        } else {
00108
         memset((char *)iov->iov_base, 0, CHUNK_SZ);
00109
00110
        return SUCCESS_CODE;
00111 }
```

#### 5.1.3.5 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,out	req	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
```

Remaining bytes of the message (content + header + foot) stored in the current IOVEC

```
Definition at line 350 of file low_saurion.c.
00350
00351
          // Initial checks
00352
          if (reg->iovec count == 0) {
00353
           return ERROR_CODE;
00354
00355
00356
         // Initizalization
         size_t max_iov_cont = 0; //< Total size of request</pre>
00357
          for (size_t i = 0; i < req->iovec_count; ++i) {
00358
00359
           max_iov_cont += req->iov[i].iov_len;
00360
         size_t cont_sz = 0;
size_t cont_rem = 0;
size_t curr_iov = 0;
size_t curr_iov_off = 0;
size_t dest_off = 0;
void *dest_ptr = NULL;
//< Message content size
//< Remaining bytes of message content
size tourrentive reading
size_t curr_iov_off = 0;
//< Offset in bytes of the current IOVEC
size_t dest_off = 0;
//< Write offset on the destiny array
void *dest_ptr = NULL;
//< Destiny pointer, could be dest or prev</pre>
00361
00362
00363
00364
00365
00366
00367
          if (req->prev && req->prev_size && req->prev_remain) {
          // There's a previous unfinished message
00368
00369
           cont_sz = req->prev_size;
           cont_rem = req->prev_remain;
00370
           curr_iov = 0;
00371
00372
            curr_iov_off = 0;
00373
            dest_off = cont_sz - cont_rem;
00374
           if (cont_rem <= max_iov_cont) {</pre>
00375
              *dest = req->prev;
             dest_ptr = *dest;
00376
             req->prev = NULL;
req->prev_size = 0;
00377
00378
00379
              req->prev_remain = 0;
00380
00381
              dest_ptr = req->prev;
              *dest = NULL;
00382
00383
00384
         } else if (req->next iov || req->next offset) {
          // Reading the next message
00385
00386
            curr_iov = req->next_iov;
00387
            curr_iov_off = req->next_offset;
00388
            cont_sz = *((size_t *)(req->iov[curr_iov].iov_base + curr_iov_off));
            cont_sz = ntohll(cont_sz);
00389
            curr_iov_off += sizeof(uint64_t);
00390
00391
            cont_rem = cont_sz;
            dest_off = cont_sz - cont_rem;
00392
00393
            if ((curr_iov_off + cont_rem + 1) <= max_iov_cont) {</pre>
00394
             *dest = malloc(cont_sz);
00395
               dest_ptr = *dest;
00396
            } else {
00397
              req->prev = malloc(cont_sz);
00398
              dest_ptr = req->prev;
```

```
00399
             *dest = NULL;
00400
             *len = 0;
00401
00402
        } else {
          // Reading the first message
00403
00404
           curr iov = 0:
           curr_iov_off = 0;
00406
           cont_sz = *((size_t *)(req->iov[curr_iov].iov_base + curr_iov_off));
           cont_sz = ntohll(cont_sz);
00407
00408
           curr_iov_off += sizeof(uint64_t);
          cont_rem = cont_sz;
dest_off = cont_sz - cont_rem;
if (cont_rem <= max_iov_cont) {</pre>
00409
00410
00411
00412
             *dest = malloc(cont_sz);
00413
             dest_ptr = *dest;
00414
           } else {
            req->prev = malloc(cont_sz);
dest_ptr = req->prev;
00415
00416
             *dest = NULL;
00417
00418
          }
00419
00421
        size_t curr_iov_msg_rem = 0;
00422
        // Copy loop
uint8_t ok = 1UL;
00423
00424
        while (1) {
00425
00426
           curr_iov_msg_rem = MIN(cont_rem, (req->iov[curr_iov].iov_len - curr_iov_off));
00427
           memopy(dest_ptr + dest_off, req->iov[curr_iov].iov_base + curr_iov_off, curr_iov_msg_rem);
00428
           dest_off += curr_iov_msg_rem;
           curr_iov_off += curr_iov_msg_rem;
00429
00430
           cont_rem -= curr_iov_msg_rem;
00431
           if (cont_rem <= 0) {</pre>
00432
            // Finish reading
00433
             if (*((uint8_t *)(req->iov[curr_iov].iov_base + curr_iov_off)) != 0) {
00434
              ok = OUL;
00435
00436
             *len = cont sz;
             ++curr_iov_off;
00437
00438
             break;
00439
           if (curr_iov_off >= (req->iov[curr_iov].iov_len)) {
00440
00441
             ++curr_iov;
00442
             if (curr_iov == req->iovec_count) {
00443
               break;
00444
00445
             curr_iov_off = 0;
00446
          }
00447
        }
00448
        // Update status
00449
00450
        if (req->prev) {
00451
          req->prev_size = cont_sz;
00452
           req->prev_remain = cont_rem;
00453
          *dest = NULL;
len = 0;
00454
00455
        } else {
00456
          req->prev_size = 0;
00457
          req->prev_remain = 0;
00458
00459
         if (curr_iov < req->iovec_count) {
          uint64_t next_sz = *(uint64_t *)(req->iov[curr_iov].iov_base + curr_iov_off);
if ((req->iov[curr_iov].iov_len > curr_iov_off) && next_sz) {
00460
00461
00462
             req->next_iov = curr_iov;
00463
             req->next_offset = curr_iov_off;
00464
           } else {
00465
             req->next_iov = 0;
00466
             req->next_offset = 0;
00467
          }
00468
00469
00470
         // Finish
00471
         if (!ok) {
          \ensuremath{//} Esto solo es posible si no se encuentra un 0 al final del la lectura
00472
           // buscar el siguiente 0 y ... probar fortuna
00473
           free(dest_ptr);
dest_ptr = NULL;
00474
00475
00476
           *dest = NULL;
00477
           *len = 0;
00478
           req->next_iov = 0;
00479
           req->next_offset = 0;
00480
           for (size_t i = curr_iov; i < req->iovec_count; ++i) {
             for (size_t j = curr_iov_off; j < req->iov[i].iov_len; ++j) {
00481
00482
               uint8_t foot = *(uint8_t *)(req->iov[i].iov_base + j);
00483
               if (foot == 0) {
                 req->next_iov = i;
req->next_offset = (j + 1) % req->iov[i].iov_len;
00484
00485
00486
                 return ERROR_CODE;
```

#### 5.1.3.6 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 585 of file low\_saurion.c.

```
00585
00586
        // Asignar memoria
00587
        struct saurion *p = (struct saurion *)malloc(sizeof(struct saurion));
00588
        if (!p) {
00589
          return NULL;
00590
        // Inicializar mutex
00591
00592
        int ret = 0;
        ret = pthread_mutex_init(&p->status_m, NULL);
00593
00594
        if (ret) {
00595
        free(p);
00596
          return NULL;
00597
00598
        ret = pthread_cond_init(&p->status_c, NULL);
00599
        if (ret) {
00600
        free(p);
00601
00602
00603
        p\text{->}m\_rings = (pthread\_mutex\_t *) \\ malloc(n\_threads * sizeof(pthread\_mutex\_t));
        if (!p->m_rings) {
00604
00605
         free(p);
00606
          return NULL;
00607
00608
        for (uint32_t i = 0; i < n_threads; ++i) {</pre>
00609
         pthread_mutex_init(&(p->m_rings[i]), NULL);
00610
        // Inicializar miembros
00611
00612
        p->ss = 0;
00613
        n_threads = (n_threads < 2 ? 2 : n_threads);</pre>
00614
        n_threads = (n_threads > NUM_CORES ? NUM_CORES : n_threads);
00615
        p->n_threads = n_threads;
00616
        p->status = 0;
        p->list = NULL;
00617
00618
       p->cb.on_connected = NULL;
00619
        p->cb.on_connected_arg = NULL;
00620
       p->cb.on_readed = NULL;
00621
       p->cb.on_readed_arg = NULL;
00622
       p->cb.on_wrote = NULL;
00623
       p->cb.on_wrote_arg = NULL;
00624 p->cb.on_closed = NULL;
00625 p->cb.on_closed_arg = NULL;
```

```
p->cb.on_error = NULL;
00627
        p->cb.on_error_arg = NULL;
00628
        p->next = 0;
        // Inicializar efds
00629
00630
        p->efds = (int *)malloc(sizeof(int) * p->n_threads);
        if (!p->efds) {
00631
00632
         free(p->m_rings);
00633
          free(p);
00634
          return NULL;
00635
00636
        for (uint32_t i = 0; i < p->n_threads; ++i) {
          p->efds[i] = eventfd(0, EFD_NONBLOCK);
00637
           if (p->efds[i] == ERROR_CODE) {
  for (uint32_t j = 0; j < i; ++j) {</pre>
00638
00639
00640
              close(p->efds[j]);
00641
             free(p->efds);
00642
00643
             free(p->m_rings);
00644
             free(p);
00645
             return NULL;
00646
00647
        // Inicializar rings
00648
        p->rings = (struct io_uring *)malloc(sizeof(struct io_uring) * p->n_threads);
00649
        if (!p->rings) {
    for (uint32_t j = 0; j < p->n_threads; ++j) {
00650
00652
            close(p->efds[j]);
00653
00654
          free(p->efds);
          free(p->m_rings);
00655
00656
          free(p);
00657
          return NULL;
00658
00659
        for (uint32_t i = 0; i < p->n_threads; ++i) {
          memset(&p->rings[i], 0, sizeof(struct io_uring));
ret = io_uring_queue_init(SAURION_RING_SIZE, &p->rings[i], 0);
00660
00661
00662
           if (ret) {
            for (uint32_t j = 0; j < p->n_threads; ++j) {
00663
00664
              close(p->efds[j]);
00665
00666
             free(p->efds);
             free(p->rings);
00667
             free (p->m_rings);
00668
00669
             free(p);
00670
             return NULL;
00671
00672
00673
        p->pool = ThreadPool_create(p->n_threads);
00674
        return p;
00675 }
```

#### 5.1.3.7 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.

#### 

```
pthread_mutex_unlock(&s->status_m);
        ThreadPool_destroy(s->pool);
for (uint32_t i = 0; i < s->n_threads; ++i) {
00839
       io_uring_queue_exit(&s->rings[i]);
00840
00841
         pthread_mutex_destroy(&s->m_rings[i]);
00842
00843
       free(s->m_rings);
00844
        list_free(&s->list);
00845
        for (uint32_t i = 0; i < s->n_threads; ++i) {
00846
         close(s->efds[i]);
00847
00848
       free(s->efds);
       if (!s->ss) {
00849
00850
         close(s->ss);
00851
00852
       free(s->rings);
        pthread_mutex_destroy(&s->status_m);
00853
00854
        pthread_cond_destroy(&s->status_c);
        free(s);
00856 }
```

#### 5.1.3.8 saurion\_send()

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

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 858 of file low_saurion.c.

00858

00859 add_write(s, fd, msg, next(s));

00860 }
```

#### 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 808 of file low\_saurion.c.

```
00809
        pthread_mutex_init(&print_mutex, NULL);
         ThreadPool_init(s->pool);
00811
        ThreadPool_add_default(s->pool, saurion_worker_master, s);
        struct saurion_wrapper *ss = NULL;
for (uint32_t i = 1; i < s->n_threads; ++i) {
00812
00813
00814
          ss = (struct saurion_wrapper *)malloc(sizeof(struct saurion_wrapper));
00815
          ss->s = s;
          ss->sel = i;
00816
00817
          ThreadPool_add_default(s->pool, saurion_worker_slave, ss);
00818
        return SUCCESS_CODE;
00819
00820 }
```

#### 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 822 of file low\_saurion.c.

```
00822
00823
00824
00825
00825

while (write(s->efds[i], &u, sizeof(u)) < 0) {
    usleep(TIMEOUT_RETRY);
    0828
    }
00828
}
ThreadPool_wait_empty(s->pool);
```

#### 5.1.3.11 set request()

```
int set_request (
          struct request ** r,
           struct Node ** l,
           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 131 of file low saurion.c.

```
00131
00132
        uint64_t full_size = s;
00133
00134
          full_size += (sizeof(uint64_t) + 1);
00135
        size_t amount = full_size / CHUNK_SZ;
00136
00137
        amount = amount + (full_size % CHUNK_SZ == 0 ? 0 : 1);
00138
        struct request *temp =
00139
             (struct request *)malloc(sizeof(struct request) + sizeof(struct iovec) * amount);
00140
        if (!temp) {
00141
          return ERROR_CODE;
00142
00143
        if (!*r) {
00144
          *r = temp;
           (*r)->prev = NULL;
00145
00146
           (*r)->prev_size = 0;
00147
           (*r)->prev_remain = 0;
00148
           (*r) \rightarrow next_iov = 0;
00149
           (*r) \rightarrow next_offset = 0;
00150
        } else {
00151
          temp->client_socket = (*r)->client_socket;
00152
          temp->event_type = (*r)->event_type;
00153
          temp->prev = (*r)->prev;
          temp->prev_size = (*r)->prev_size;
temp->prev_remain = (*r)->prev_remain;
00154
00155
00156
          temp->next_iov = (*r)->next_iov;
00157
          temp->next_offset = (*r)->next_offset;
```

```
*r = temp;
00158
00159
           struct request *req = *r;
req->iovec_count = (int)amount;
00160
00161
           void **children_ptr = (void **)malloc(amount * sizeof(void *));
if (!children_ptr) {
  free_request(req, children_ptr, 0);
00162
00163
00164
00165
              return ERROR_CODE;
00166
           for (size_t i = 0; i < amount; ++i) {
  if (!allocate_iovec(&req->iov[i], amount, i, full_size, children_ptr)) {
    free_request(req, children_ptr, amount);
    return ERROR_CODE;
00167
00168
00169
00170
00171
00172
              if (!initialize_iovec(&req->iov[i], amount, i, m, s, h)) {
00173
00174
                free_request(req, children_ptr, amount);
return ERROR_CODE;
00175
              }
00176
           if (list_insert(1, req, amount, children_ptr)) {
   free_request(req, children_ptr, amount);
00177
00178
00179
              return ERROR_CODE;
00180
           free(children_ptr);
return SUCCESS_CODE;
00181
00182
00183 }
```

### **Class Documentation**

### 6.1 ThreadPool::AsyncMultiQueue Struct Reference

#### **Public Member Functions**

- AsyncMultiQueue ()
- ∼AsyncMultiQueue ()
- AsyncMultiQueue (const AsyncMultiQueue &)=delete
- AsyncMultiQueue & operator= (const AsyncMultiQueue &)=delete
- AsyncMultiQueue (AsyncMultiQueue &&)=delete
- AsyncMultiQueue & operator= (AsyncMultiQueue &&)=delete
- void new\_queue (uint32\_t qid, uint32\_t cnt)
- void remove\_queue (uint32\_t qid)
- void push (uint32\_t qid, void(\*nfn)(void \*), void \*arg)
- Task \* front (uint32\_t &qid)
- void pop (uint32 t qid)
- void clear ()
- · bool empty ()

#### **Private Attributes**

- std::unordered\_map< uint32\_t, AsyncQueue \* > m\_queues
- std::unordered\_map< uint32\_t, AsyncQueue \* >::iterator m\_it

#### 6.1.1 Detailed Description

Definition at line 50 of file threadpool.hpp.

#### 6.1.2 Constructor & Destructor Documentation

26 Class Documentation

#### 6.1.2.1 AsyncMultiQueue() [1/3]

```
TP::AsyncMultiQueue::AsyncMultiQueue ( ) [explicit]
```

#### 6.1.2.2 ∼AsyncMultiQueue()

```
TP::AsyncMultiQueue::~AsyncMultiQueue ( )
```

# Definition at line 56 of file threadpool.cpp. 00056 00057 00058 00058 00059 00059 00060 } for (auto& queue: m\_queues) { delete queue.second; 00060

#### 6.1.2.3 AsyncMultiQueue() [2/3]

#### 6.1.2.4 AsyncMultiQueue() [3/3]

#### 6.1.3 Member Function Documentation

#### 6.1.3.1 clear()

```
void TP::AsyncMultiQueue::clear ( )
```

#### Definition at line 101 of file threadpool.cpp.

```
00101
           if (empty()) {
00102
         return;
}
00103
00104
00105
          auto newit = m_queues.begin();
00106 while (newit != m_queues.end()) {
00107 while (!newit->second->empty()) {
00108 delete newit->second->front();
00109
                newit->second->pop();
00110
00111
              ++newit;
00112
          }
00113 }
```

#### 6.1.3.2 empty()

#### 6.1.3.3 front()

Definition at line 80 of file threadpool.cpp.

```
08000
00081
        if (empty()) {
00082
         throw std::out_of_range("empty queue");
00083
00084
       auto newit = m_it;
00085
        ++newit;
00086
       while (newit != m_it) {
00087
        if (newit == m_queues.end()) {
00088
           newit = m_queues.begin();
00089
00090
         if (!newit->second->empty()) {
00091
          break;
00092
00093
          ++newit;
00094
00095
       Task* task = newit->second->front();
       qid = newit->first;
00096
       m_it = newit;
00098
       return task;
00099 }
```

# 6.1.3.4 new\_queue()

# Definition at line 62 of file threadpool.cpp.

#### 6.1.3.5 operator=() [1/2]

#### 6.1.3.6 operator=() [2/2]

# 6.1.3.7 pop()

#### Definition at line 100 of file threadpool.cpp.

```
00100 { m_queues.at(qid)->pop(); }
```

#### 6.1.3.8 push()

#### Definition at line 77 of file threadpool.cpp.

```
00077
00078  m_queues.at(qid)->push(new Task{nfn, arg});
00079 }
```

#### 6.1.3.9 remove\_queue()

#### Definition at line 68 of file threadpool.cpp.

```
00068
00069     auto queue = m_queues.find(qid);
00070     if (queue == m_queues.end()) {
        throw std::out_of_range("queue not found");
00072     }
00073     delete queue->second;
00074     m_queues.erase(qid);
00075 }
```

#### 6.1.4 Member Data Documentation

#### 6.1.4.1 m\_it

std::unordered\_map<uint32\_t,AsyncQueue\*>::iterator ThreadPool::AsyncMultiQueue::m\_it [private]

Definition at line 53 of file threadpool.hpp.

#### 6.1.4.2 m\_queues

```
std::unordered_map<uint32_t, AsyncQueue*> ThreadPool::AsyncMultiQueue::m_queues [private]
```

Definition at line 52 of file threadpool.hpp.

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

- /\_\_w/saurion/saurion/include/threadpool.hpp
- /\_\_w/saurion/saurion/src/threadpool.cpp

# 6.2 ThreadPool::AsyncQueue Struct Reference

#### **Public Member Functions**

- AsyncQueue (uint32\_t cnt)
- ∼AsyncQueue ()
- AsyncQueue (const AsyncQueue &)=delete
- AsyncQueue & operator= (const AsyncQueue &)=delete
- AsyncQueue (AsyncQueue &&)=delete
- AsyncQueue & operator= (AsyncQueue &&)=delete
- void push (Task \*task)
- Task \* front ()
- void pop ()
- bool empty ()

#### **Private Attributes**

- $std::queue < Task * > m_queue$
- uint32 t m max
- uint32\_t m\_cnt
- pthread\_mutex\_t m\_mtx = PTHREAD\_MUTEX\_INITIALIZER

# 6.2.1 Detailed Description

Definition at line 29 of file threadpool.hpp.

#### 6.2.2 Constructor & Destructor Documentation

#### 6.2.2.1 AsyncQueue() [1/3]

#### 6.2.2.2 ~AsyncQueue()

```
TP::AsyncQueue::~AsyncQueue ( )

Definition at line 18 of file threadpool.cpp.

00018 { pthread_mutex_destroy(&m_mtx); }
```

# 6.2.2.3 AsyncQueue() [2/3]

#### 6.2.2.4 AsyncQueue() [3/3]

# 6.2.3 Member Function Documentation

#### 6.2.3.1 empty()

```
bool TP::AsyncQueue::empty ( )
```

# Definition at line 43 of file threadpool.cpp.

```
00043 {
00044 pthread_mutex_lock(&m_mtx);
00045 bool empty = m_queue.empty();
00046 pthread_mutex_unlock(&m_mtx);
00047 return empty;
```

#### 6.2.3.2 front()

```
Task * TP::AsyncQueue::front ( )
Definition at line 25 of file threadpool.cpp.
00025
00026
        pthread_mutex_lock(&m_mtx);
00027
        if ((m_cnt >= m_max) && (m_max != 0)) {
00028
       pthread_mutex_unlock(&m_mtx);
00029
          throw std::out_of_range("reached max parallel tasks");
00030
       Task* task = m_queue.front();
00031
       m_queue.pop();
++m_cnt;
00032
00033
00034
       pthread_mutex_unlock(&m_mtx);
00035
       return task;
00036 }
```

## 6.2.3.3 operator=() [1/2]

#### 6.2.3.4 operator=() [2/2]

#### 6.2.3.5 pop()

```
void TP::AsyncQueue::pop ( )
```

#### Definition at line 37 of file threadpool.cpp.

```
00037
00038 pthread_mutex_lock(&m_mtx);
00039 --m_cnt;
00040 pthread_mutex_unlock(&m_mtx);
00041 }
```

#### 6.2.3.6 push()

## Definition at line 20 of file threadpool.cpp.

# 6.2.4 Member Data Documentation

# 6.2.4.1 m\_cnt

```
uint32_t ThreadPool::AsyncQueue::m_cnt [private]
```

Definition at line 33 of file threadpool.hpp.

#### 6.2.4.2 m\_max

```
uint32_t ThreadPool::AsyncQueue::m_max [private]
```

Definition at line 32 of file threadpool.hpp.

#### 6.2.4.3 m mtx

```
pthread_mutex_t ThreadPool::AsyncQueue::m_mtx = PTHREAD_MUTEX_INITIALIZER [private]
```

Definition at line 34 of file threadpool.hpp.

# 6.2.4.4 m\_queue

```
std::queue<Task*> ThreadPool::AsyncQueue::m_queue [private]
```

Definition at line 31 of file threadpool.hpp.

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

- /\_\_w/saurion/saurion/include/threadpool.hpp
- /\_\_w/saurion/saurion/src/threadpool.cpp

# 6.3 Node Struct Reference

Collaboration diagram for Node:

6.3 Node Struct Reference 33

# **Public Attributes**

- void \* ptr
- size\_t size
- struct Node \*\* children
- struct Node \* next

# 6.3.1 Detailed Description

Definition at line 6 of file linked\_list.c.

# 6.3.2 Member Data Documentation

#### 6.3.2.1 children

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

Definition at line 9 of file linked\_list.c.

#### 6.3.2.2 next

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

Definition at line 10 of file linked\_list.c.

#### 6.3.2.3 ptr

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

Definition at line 7 of file linked\_list.c.

#### 6.3.2.4 size

```
size_t Node::size
```

Definition at line 8 of file linked\_list.c.

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

/\_\_w/saurion/saurion/src/linked\_list.c

# 6.4 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 []

# 6.4.1 Detailed Description

Definition at line 15 of file low\_saurion.c.

#### 6.4.2 Member Data Documentation

#### 6.4.2.1 client\_socket

```
int request::client_socket
```

Definition at line 23 of file low\_saurion.c.

# 6.4.2.2 event\_type

```
int request::event_type
```

Definition at line 21 of file low\_saurion.c.

### 6.4.2.3 iov

struct iovec request::iov[]

Definition at line 24 of file low\_saurion.c.

#### 6.4.2.4 iovec\_count

```
size_t request::iovec_count
```

Definition at line 22 of file low\_saurion.c.

#### 6.4.2.5 next\_iov

```
size_t request::next_iov
```

Definition at line 19 of file low\_saurion.c.

#### 6.4.2.6 next\_offset

```
size_t request::next_offset
```

Definition at line 20 of file low\_saurion.c.

### 6.4.2.7 prev

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

Definition at line 16 of file low\_saurion.c.

#### 6.4.2.8 prev\_remain

```
size_t request::prev_remain
```

Definition at line 18 of file low\_saurion.c.

# 6.4.2.9 prev\_size

```
size_t request::prev_size
```

Definition at line 17 of file low\_saurion.c.

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

• /\_\_w/saurion/saurion/src/low\_saurion.c

# 6.5 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
- ThreadPool \* pool
- uint32\_t n\_threads
- uint32\_t next

# 6.5.1 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 137 of file low\_saurion.h.

#### 6.5.2 Member Data Documentation

#### 6.5.2.1 efds

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

Eventfd descriptors used for internal signaling between threads.

Definition at line 142 of file low\_saurion.h.

#### 6.5.2.2 list

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

Linked list for storing active requests.

Definition at line 143 of file low saurion.h.

# 6.5.2.3 m\_rings

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

Array of mutexes to protect the io\_uring rings during concurrent access.

Definition at line 140 of file low\_saurion.h.

#### 6.5.2.4 n threads

```
uint32_t saurion::n_threads
```

Number of threads in the thread pool.

Definition at line 148 of file low\_saurion.h.

# 6.5.2.5 next

```
uint32_t saurion::next
```

Index of the next io\_uring ring to which an event will be added.

Definition at line 149 of file low\_saurion.h.

#### 6.5.2.6 pool

```
ThreadPool* saurion::pool
```

Thread pool for executing tasks in parallel.

Definition at line 147 of file low\_saurion.h.

#### 6.5.2.7 rings

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

Array of io\_uring structures for managing the event queue.

Definition at line 138 of file low\_saurion.h.

# 6.5.2.8 ss

```
int saurion::ss
```

Server socket descriptor for accepting connections.

Definition at line 141 of file low\_saurion.h.

#### 6.5.2.9 status

```
int saurion::status
```

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

Definition at line 146 of file low saurion.h.

#### 6.5.2.10 status\_c

```
pthread_cond_t saurion::status_c
```

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

Definition at line 145 of file low\_saurion.h.

## 6.5.2.11 status\_m

```
pthread_mutex_t saurion::status_m
```

Mutex to protect the state of the structure.

Definition at line 144 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 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 () 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

#### 6.6.1 Detailed Description

Definition at line 6 of file saurion.hpp.

#### 6.6.2 Member Typedef Documentation

#### 6.6.2.1 ClosedCb

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

Definition at line 11 of file saurion.hpp.

#### 6.6.2.2 ConnectedCb

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

Definition at line 8 of file saurion.hpp.

#### 6.6.2.3 ErrorCb

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

Definition at line 12 of file saurion.hpp.

#### 6.6.2.4 ReadedCb

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

Definition at line 9 of file saurion.hpp.

#### 6.6.2.5 WroteCb

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

Definition at line 10 of file saurion.hpp.

# 6.6.3 Constructor & Destructor Documentation

# 6.6.3.1 Saurion() [1/3]

#### Definition at line 7 of file saurion.cpp.

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

#### 6.6.3.2 ~Saurion()

```
Saurion::~Saurion ()

Definition at line 15 of file saurion.cpp.

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

#### 6.6.3.3 Saurion() [2/3]

#### 6.6.3.4 Saurion() [3/3]

# 6.6.4 Member Function Documentation

# 6.6.4.1 init()

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

# Definition at line 17 of file saurion.cpp.

```
00017
00018    if (!saurion_start(this->s)) {
00019       return;
00020    }
00021 }
```

#### 6.6.4.2 on\_closed()

{

#### Definition at line 43 of file saurion.cpp.

```
00043

00044 s->cb.on_closed = ncb;

00045 s->cb.on_closed_arg = arg;

00046 return this;

00047 }
```

{

{

#### 6.6.4.3 on\_connected()

#### Definition at line 25 of file saurion.cpp.

```
00025

00026 s->cb.on_connected = ncb;

00027 s->cb.on_connected_arg = arg;

00028 return this;

00029 }
```

#### 6.6.4.4 on\_error()

#### Definition at line 49 of file saurion.cpp.

```
00049
00050 s->cb.on_error = ncb;
00051 s->cb.on_error_arg = arg;
00052 return this;
00053 }
```

# 6.6.4.5 on\_readed()

#### Definition at line 31 of file saurion.cpp.

```
00031
00032 s->cb.on_readed = ncb;
00033 s->cb.on_readed_arg = arg;
00034 return this;
00035 }
```

#### 6.6.4.6 on\_wrote()

#### Definition at line 37 of file saurion.cpp.

#### 6.6.4.7 operator=() [1/2]

# 6.6.4.8 operator=() [2/2]

#### 6.6.4.9 send()

# Definition at line 55 of file saurion.cpp.

```
00055 { saurion_send(this->s, fd, msg); }
```

## 6.6.4.10 stop()

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

# Definition at line 23 of file saurion.cpp.

00023 { saurion\_stop(this->s); }

#### 6.6.5 Member Data Documentation

## 6.6.5.1 s

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

Definition at line 34 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

# 6.7 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

#### 6.7.1 Detailed Description

Structure containing callback functions to handle socket events.

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

Definition at line 159 of file low saurion.h.

# 6.7.2 Member Data Documentation

#### 6.7.2.1 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 195 of file low\_saurion.h.

#### 6.7.2.2 on\_closed\_arg

void\* saurion::saurion\_callbacks::on\_closed\_arg

Additional argument for the close callback.

Definition at line 196 of file low\_saurion.h.

#### 6.7.2.3 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.

Definition at line 166 of file low\_saurion.h.

# 6.7.2.4 on\_connected\_arg

void\* saurion::saurion\_callbacks::on\_connected\_arg

Additional argument for the connection callback.

Definition at line 167 of file low\_saurion.h.

#### 6.7.2.5 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.
	Pointer to the error message.
Generated by Doxygen Length of the error message.	
arg	Additional user-provided argument.

Definition at line 206 of file low\_saurion.h.

#### 6.7.2.6 on\_error\_arg

void\* saurion::saurion\_callbacks::on\_error\_arg

Additional argument for the error callback.

Definition at line 207 of file low\_saurion.h.

#### 6.7.2.7 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 177 of file low\_saurion.h.

#### 6.7.2.8 on readed arg

void\* saurion::saurion\_callbacks::on\_readed\_arg

Additional argument for the read callback.

Definition at line 178 of file low\_saurion.h.

#### 6.7.2.9 on\_wrote

void(\* saurion::saurion\_callbacks::on\_wrote) (const int fd, void \*arg)

Callback for handling write events.

#### **Parameters**

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

Definition at line 186 of file low\_saurion.h.

#### 6.7.2.10 on\_wrote\_arg

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

Additional argument for the write callback.

Definition at line 187 of file low\_saurion.h.

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

• /\_\_w/saurion/saurion/include/low\_saurion.h

# 6.8 saurion\_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

# 6.8.1 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 21 of file low\_saurion.h.

#### 6.8.2 Member Data Documentation

#### 6.8.2.1 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 57 of file low\_saurion.h.

## 6.8.2.2 on\_closed\_arg

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

Additional argument for the close callback.

Definition at line 58 of file low\_saurion.h.

## 6.8.2.3 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 28 of file low\_saurion.h.

#### 6.8.2.4 on\_connected\_arg

void\* saurion\_callbacks::on\_connected\_arg

Additional argument for the connection callback.

Definition at line 29 of file low\_saurion.h.

#### 6.8.2.5 on\_error

 $\label{local_const_con$ 

Callback for handling error events.

#### **Parameters**

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

Definition at line 68 of file low saurion.h.

#### 6.8.2.6 on\_error\_arg

void\* saurion\_callbacks::on\_error\_arg

Additional argument for the error callback.

Definition at line 69 of file low\_saurion.h.

# 6.8.2.7 on\_readed

 $void(* saurion\_callbacks::on\_readed)$  (const int fd, const void \*const content, const  $ssize\_\leftarrow$ 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 39 of file low\_saurion.h.

# 6.8.2.8 on\_readed\_arg

void\* saurion\_callbacks::on\_readed\_arg

Additional argument for the read callback.

Definition at line 40 of file low\_saurion.h.

#### 6.8.2.9 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 48 of file low\_saurion.h.

# 6.8.2.10 on\_wrote\_arg

void\* saurion\_callbacks::on\_wrote\_arg

Additional argument for the write callback.

Definition at line 49 of file low\_saurion.h.

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

• /\_\_w/saurion/saurion/include/low\_saurion.h

# 6.9 saurion\_wrapper Struct Reference

Collaboration diagram for saurion\_wrapper:

#### **Public Attributes**

- struct saurion \* s
- uint32\_t sel

# 6.9.1 Detailed Description

Definition at line 31 of file low\_saurion.c.

#### 6.9.2 Member Data Documentation

#### 6.9.2.1 s

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

Definition at line 32 of file low\_saurion.c.

## 6.9.2.2 sel

```
uint32_t saurion_wrapper::sel
```

Definition at line 33 of file low\_saurion.c.

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

• /\_\_w/saurion/saurion/src/low\_saurion.c

# 6.10 Task Struct Reference

```
#include <threadpool.hpp>
```

#### **Public Member Functions**

- Task (void(\*nfn)(void \*), void \*narg)
- Task (const Task &)=delete
- Task (Task &&)=delete
- Task & operator= (const Task &)=delete
- Task & operator= (Task &&)=delete
- ~Task ()=default

# **Public Attributes**

```
void(* function )(void *)void * argument
```

# 6.10.1 Detailed Description

Definition at line 14 of file threadpool.hpp.

# 6.10.2 Constructor & Destructor Documentation

# 6.10.2.2 Task() [2/3]

# 6.10.2.3 Task() [3/3]

#### 6.10.2.4 ∼Task()

```
Task::~Task ( ) [default]
```

# 6.10.3 Member Function Documentation

#### 6.10.3.1 operator=() [1/2]

#### 6.10.3.2 operator=() [2/2]

#### 6.10.4 Member Data Documentation

#### 6.10.4.1 argument

```
void* Task::argument
```

Definition at line 16 of file threadpool.hpp.

#### 6.10.4.2 function

```
void(* Task::function) (void *)
```

Definition at line 15 of file threadpool.hpp.

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

- /\_\_w/saurion/saurion/include/threadpool.hpp
- /\_\_w/saurion/saurion/src/threadpool.cpp

# 6.11 ThreadPool Class Reference

```
#include <threadpool.hpp>
```

Collaboration diagram for ThreadPool:

# Classes

- struct AsyncMultiQueue
- struct AsyncQueue

#### **Public Member Functions**

- ThreadPool ()
- ThreadPool (size\_t num\_threads)
- ThreadPool (const ThreadPool &)=delete
- ThreadPool & operator= (const ThreadPool &)=delete
- ThreadPool (ThreadPool &&)=delete
- ThreadPool & operator= (ThreadPool &&)=delete
- void init ()
- void stop ()
- void add (uint32\_t qid, void(\*nfn)(void \*), void \*arg)
- void add (void(\*nfn)(void \*), void \*arg)
- void new\_queue (uint32\_t qid, uint32\_t cnt)
- void remove\_queue (uint32\_t qid)
- bool empty ()
- void wait empty ()
- ∼ThreadPool ()

# **Private Types**

- typedef struct ThreadPool::AsyncQueue AsyncQueue
- typedef struct ThreadPool::AsyncMultiQueue AsyncMultiQueue

#### **Private Member Functions**

- void wait\_closeable ()
- void thread\_worker ()

#### **Static Private Member Functions**

static void \* thread entry (void \*arg)

# **Private Attributes**

- size\_t m\_nth
- size\_t m\_started
- AsyncMultiQueue m queues
- pthread\_mutex\_t m\_q\_mtx = PTHREAD\_MUTEX\_INITIALIZER
- pthread cond t m q cond = PTHREAD COND INITIALIZER
- pthread t \* m ths
- volatile sig\_atomic\_t m\_fstop
- volatile sig\_atomic\_t m\_faccept

# **Static Private Attributes**

• static pthread mutex ts mtx = PTHREAD MUTEX INITIALIZER

# 6.11.1 Detailed Description

Definition at line 27 of file threadpool.hpp.

# 6.11.2 Member Typedef Documentation

# 6.11.2.1 AsyncMultiQueue

```
typedef struct ThreadPool::AsyncMultiQueue ThreadPool::AsyncMultiQueue [private]
```

## 6.11.2.2 AsyncQueue

```
typedef struct ThreadPool::AsyncQueue ThreadPool::AsyncQueue [private]
```

#### 6.11.3 Constructor & Destructor Documentation

# 6.11.3.1 ThreadPool() [1/4]

```
TP::ThreadPool ( )
Definition at line 128 of file threadpool.cpp.
00128 : ThreadPool(4) {}
```

# 6.11.3.2 ThreadPool() [2/4]

```
TP::ThreadPool (
           size_t num_threads ) [explicit]
Definition at line 130 of file threadpool.cpp.
m_ths(new pthread_t[m_nth]{0}),
m_fstop(0),
m_faccept(0) {}
00133
```

00134 00135

#### 6.11.3.3 ThreadPool() [3/4]

#### 6.11.3.4 ThreadPool() [4/4]

#### 6.11.3.5 ∼ThreadPool()

```
TP::\sim ThreadPool ()
```

#### Definition at line 233 of file threadpool.cpp.

# 6.11.4 Member Function Documentation

#### 6.11.4.1 add() [1/2]

#### Definition at line 169 of file threadpool.cpp.

```
00169
00170
        if (m_faccept == 0) {
        throw std::logic_error("threadpool already closed");
00171
00172
       if (nfn == nullptr) {
       throw std::logic_error("function pointer cannot be null");
}
00174
00175
00176
       bool failed = false;
00177
       pthread_mutex_lock(&m_q_mtx);
00178
       try {
00179
         m_queues.push(qid, nfn, arg);
00180
       } catch (const std::out_of_range& e) {
       failed = true;
}
00181
00182
00183
       pthread_cond_signal(&m_q_cond);
00184
       pthread_mutex_unlock(&m_q_mtx);
       if (failed) {
00185
00186
         throw std::out_of_range("queue not found");
00187 }
00188 }
```

#### 6.11.4.2 add() [2/2]

# 6.11.4.3 empty()

```
bool TP::empty ( )
```

Definition at line 218 of file threadpool.cpp.

```
00218 { return m_queues.empty(); }
```

#### 6.11.4.4 init()

```
void TP::init ( )
```

Definition at line 137 of file threadpool.cpp.

#### 6.11.4.5 new\_queue()

Definition at line 192 of file threadpool.cpp.

```
00192
00193
pthread_mutex_lock(&m_q_mtx);
try {
00195
    m_queues.new_queue(qid, cnt);
00196
} catch (const std::out_of_range& e) {
00197
    pthread_mutex_unlock(&m_q_mtx);
00198
    trow e;
00199
} pthread_mutex_unlock(&m_q_mtx);
00200
pthread_mutex_unlock(&m_q_mtx);
```

#### 6.11.4.6 operator=() [1/2]

#### 6.11.4.7 operator=() [2/2]

#### 6.11.4.8 remove\_queue()

# Definition at line 202 of file threadpool.cpp.

```
00202
00203
        if (qid != 0) {
00204
          bool failed = false;
00205
          pthread_mutex_lock(&m_q_mtx);
00206
00207
           m_queues.remove_queue(qid);
         } catch (const std::out_of_range& e) {
  failed = true;
00208
00209
00210
00211
          pthread_cond_signal(&m_q_cond);
00212
          pthread_mutex_unlock(&m_q_mtx);
00213
          if (failed) {
00214
            throw std::out_of_range("queue not found");
00215
00216
00217 }
```

#### 6.11.4.9 stop()

```
void TP::stop ( )
```

# Definition at line 150 of file threadpool.cpp.

```
00150
00151
         if (m_started == 0) {
        ,m_sta:
    return;
}
00152
00153
       m_fstop = 0;
m_faccept = 0;
00154
00155
00156
        wait_empty();
00157
00158
        pthread_mutex_lock(&m_q_mtx);
00159
        m_fstop = 1;
00160
        pthread_cond_broadcast(&m_q_cond);
00161
        pthread_mutex_unlock(&m_q_mtx);
        // Detener los hilos
for (size_t i = 0; i < m_started; ++i) {</pre>
00162
        pthread_join(m_ths[i], nullptr);
}
00163
00164
00165
00166
        m_started = 0;
00167
        m_nth = 0;
00168 }
```

#### 6.11.4.10 thread\_entry()

#### 6.11.4.11 thread\_worker()

```
void TP::thread_worker ( ) [private]
```

#### Definition at line 240 of file threadpool.cpp.

```
// Lógica del trabajador del hilo
00241
        uint32_t qid = 0;
while (m_fstop == 0) {
00242
00243
00244
         // Buscar una tarea para ejecutar
          try {
00246
            pthread_mutex_lock(&m_q_mtx);
00247
            Task* task = m_queues.front(qid);
00248
            pthread_cond_signal(&m_q_cond);
00249
            pthread_mutex_unlock(&m_q_mtx);
00250
            try {
              task->function(task->argument);
00252
            } catch (...) {
00253
          delete task; // TODO delete task
pthread_mutex_lock(&m_q_mtx);
00254
00255
00256
            m_queues.pop(qid);
          pthread_cond_broadcast(&m_q_cond);
00258
            pthread_mutex_unlock(&m_q_mtx);
00259
          } catch (const std::out_of_range& e) {
00260
            pthread_mutex_unlock(&m_q_mtx);
00261
            wait_closeable();
00262
00263
       }
00264 }
```

#### 6.11.4.12 wait\_closeable()

```
void TP::wait_closeable ( ) [private]
```

#### Definition at line 219 of file threadpool.cpp.

#### 6.11.4.13 wait\_empty()

```
void TP::wait_empty ( )
```

#### Definition at line 226 of file threadpool.cpp.

```
00227 pthread_mutex_lock(&m_q_mtx);
00228 while (!m_queues.empty()) {
00229 pthread_cond_wait(&m_q_cond, &m_q_mtx);
00230 }
00231 pthread_mutex_unlock(&m_q_mtx);
00232 }
```

# 6.11.5 Member Data Documentation

# 6.11.5.1 m\_faccept

```
volatile sig_atomic_t ThreadPool::m_faccept [private]
```

Definition at line 106 of file threadpool.hpp.

# 6.11.5.2 m\_fstop

```
volatile sig_atomic_t ThreadPool::m_fstop [private]
```

Definition at line 105 of file threadpool.hpp.

# 6.11.5.3 m\_nth

```
size_t ThreadPool::m_nth [private]
```

Definition at line 98 of file threadpool.hpp.

# 6.11.5.4 m\_q\_cond

```
pthread_cond_t ThreadPool::m_q_cond = PTHREAD_COND_INITIALIZER [private]
```

Definition at line 102 of file threadpool.hpp.

#### 6.11.5.5 m\_q\_mtx

```
pthread_mutex_t ThreadPool::m_q_mtx = PTHREAD_MUTEX_INITIALIZER [private]
```

Definition at line 101 of file threadpool.hpp.

#### 6.11.5.6 m\_queues

```
AsyncMultiQueue ThreadPool::m_queues [private]
```

Definition at line 100 of file threadpool.hpp.

# 6.11.5.7 m\_started

```
size_t ThreadPool::m_started [private]
```

Definition at line 99 of file threadpool.hpp.

# 6.11.5.8 m\_ths

```
pthread_t* ThreadPool::m_ths [private]
```

Definition at line 104 of file threadpool.hpp.

#### 6.11.5.9 s mtx

```
pthread_mutex_t TP::s_mtx = PTHREAD_MUTEX_INITIALIZER [static], [private]
```

Definition at line 103 of file threadpool.hpp.

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

- /\_\_w/saurion/saurion/include/threadpool.hpp
- /\_\_w/saurion/saurion/src/threadpool.cpp

# **Chapter 7**

# **File Documentation**

## 7.1 /\_w/saurion/saurion/include/cthreadpool.hpp File Reference

This graph shows which files directly or indirectly include this file:

## **Typedefs**

typedef struct ThreadPool ThreadPool

## **Functions**

- ThreadPool \* ThreadPool\_create (size\_t num\_threads)
- ThreadPool \* ThreadPool\_create\_default (void)
- void ThreadPool\_init (ThreadPool \*thp)
- void ThreadPool\_stop (ThreadPool \*thp)
- void ThreadPool\_add (ThreadPool \*thp, uint32\_t qid, void(\*nfn)(void \*), void \*arg)
- void ThreadPool\_add\_default (ThreadPool \*thp, void(\*nfn)(void \*), void \*arg)
- void ThreadPool\_new\_queue (ThreadPool \*thp, uint32\_t qid, uint32\_t cnt)
- void ThreadPool\_remove\_queue (ThreadPool \*thp, uint32\_t qid)
- bool ThreadPool\_empty (ThreadPool \*thp)
- void ThreadPool\_wait\_empty (ThreadPool \*thp)
- void ThreadPool\_destroy (ThreadPool \*thp)

## 7.1.1 Typedef Documentation

#### 7.1.1.1 ThreadPool

typedef struct ThreadPool ThreadPool

Definition at line 8 of file cthreadpool.hpp.

## 7.1.2 Function Documentation

## 7.1.2.1 ThreadPool\_add()

Definition at line 11 of file cthreadpool.cpp.

```
00011 { thp->add(qid, nfn, arg); }
```

## 7.1.2.2 ThreadPool\_add\_default()

Definition at line 13 of file cthreadpool.cpp.

```
00013 { thp->add(nfn, arg); }
```

## 7.1.2.3 ThreadPool\_create()

Definition at line 3 of file cthreadpool.cpp.

```
00003 { return new ThreadPool(num_threads); }
```

## 7.1.2.4 ThreadPool\_create\_default()

Definition at line 5 of file cthreadpool.cpp.

```
00005 { return new ThreadPool(); }
```

### 7.1.2.5 ThreadPool\_destroy()

Definition at line 23 of file cthreadpool.cpp.

```
00023 { delete thp; }
```

### 7.1.2.6 ThreadPool\_empty()

```
bool ThreadPool_empty ( {\tt ThreadPool} \ * \ thp \ )
```

Definition at line 19 of file cthreadpool.cpp.

```
00019 { return thp->empty(); }
```

## 7.1.2.7 ThreadPool\_init()

Definition at line 7 of file cthreadpool.cpp.

```
00007 { thp->init(); }
```

#### 7.1.2.8 ThreadPool\_new\_queue()

Definition at line 15 of file cthreadpool.cpp.

```
00015 { thp->new_queue(qid, cnt); }
```

## 7.1.2.9 ThreadPool\_remove\_queue()

## Definition at line 17 of file cthreadpool.cpp.

```
00017 { thp->remove_queue(qid); }
```

#### 7.1.2.10 ThreadPool\_stop()

#### 7.1.2.11 ThreadPool wait empty()

```
void ThreadPool_wait_empty ( {\tt ThreadPool} \ * \ thp \ )
```

Definition at line 21 of file cthreadpool.cpp.

00021 { thp->wait\_empty(); }

## 7.2 cthreadpool.hpp

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

```
00001 #ifndef THREADPOOL_H
00002 #define THREADPOOL_H
00003
00004 #ifdef __cplusplus
00005 #include "threadpool.hpp"
00006 extern "C" {
00007 #else
00008 typedef struct ThreadPool ThreadPool;
00009 #endif // __cplusplus
00010
00011 ThreadPool *ThreadPool_create(size_t num_threads);
00012
00013 ThreadPool *ThreadPool_create_default(void);
00014
00015 void ThreadPool_init(ThreadPool *thp);
00016
00017 void ThreadPool_stop(ThreadPool *thp);
00018
00019 void ThreadPool_add(ThreadPool *thp, uint32_t qid, void (*nfn)(void *), void *arg);
00021 void ThreadPool_add_default(ThreadPool *thp, void (*nfn)(void *), void *arg);
00022
00023 void ThreadPool_new_queue(ThreadPool *thp, uint32_t qid, uint32_t cnt);
00024
00025 void ThreadPool_remove_queue(ThreadPool *thp, uint32_t qid);
00026
00027 bool ThreadPool_empty(ThreadPool *thp);
00028
00029 void ThreadPool_wait_empty(ThreadPool *thp);
00030
00031 void ThreadPool_destroy(ThreadPool *thp);
00033 #ifdef __cplusplus
00034
00035 #endif // __cplusplus
00036
00037 #endif // !THREADPOOL_H
```

## 7.3 /\_w/saurion/saurion/include/linked\_list.h File Reference

```
#include <stddef.h>
Include dependency graph for linked_list.h:
```

7.4 linked list.h

## 7.4 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 #endif
00007
00008 #include <stddef.h>
00009
00010 struct Node;
00011
00012 int list_insert(struct Node** head, void* ptr, size_t amount, void** children);
00013
00014 void list_delete_node(struct Node** head, void* ptr);
00015
00016 void list_free(struct Node** head);
00017
00018 #ifdef __cplusplus
00019
00020 #endif
00021
00022 #endif // !LINKED LIST H
```

## 7.5 / w/saurion/saurion/include/low saurion.h File Reference

```
#include #include <pthread.h>
#include "config.h"
#include "cthreadpool.hpp"
#include "linked_list.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 PACKING\_SZ 128

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

#### **Functions**

- int EXTERNAL\_set\_socket (int p)
- 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
- ThreadPool \* pool
- uint32\_t n\_threads
- uint32\_t next

## 7.5.1 Variable Documentation

#### 7.5.1.1 efds

int\* efds

Eventfd descriptors used for internal signaling between threads.

Definition at line 4 of file low\_saurion.h.

## 7.5.1.2 list

struct Node\* list

Linked list for storing active requests.

Definition at line 5 of file low\_saurion.h.

## 7.5.1.3 m\_rings

```
{\tt pthread\_mutex\_t*} \ {\tt m\_rings}
```

Array of mutexes to protect the io\_uring rings during concurrent access.

Definition at line 2 of file low saurion.h.

## 7.5.1.4 n\_threads

```
uint32_t n_threads
```

Number of threads in the thread pool.

Definition at line 10 of file low\_saurion.h.

#### 7.5.1.5 next

```
uint32_t next
```

Index of the next io\_uring ring to which an event will be added.

Definition at line 11 of file low\_saurion.h.

## 7.5.1.6 on\_closed

Callback for handling socket closures.

#### **Parameters**

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

Definition at line 35 of file low\_saurion.h.

## 7.5.1.7 on\_closed\_arg

```
void * on_closed_arg
```

Additional argument for the close callback.

Definition at line 36 of file low\_saurion.h.

### 7.5.1.8 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.

## 7.5.1.9 on\_connected\_arg

```
void * on_connected_arg
```

Additional argument for the connection callback.

Definition at line 7 of file low\_saurion.h.

## 7.5.1.10 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 46 of file low\_saurion.h.

#### 7.5.1.11 on\_error\_arg

```
void * on_error_arg
```

Additional argument for the error callback.

Definition at line 47 of file low\_saurion.h.

### 7.5.1.12 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 17 of file low\_saurion.h.

#### 7.5.1.13 on\_readed\_arg

```
void * on_readed_arg
```

Additional argument for the read callback.

Definition at line 18 of file low\_saurion.h.

#### 7.5.1.14 on\_wrote

Callback for handling write events.

### **Parameters**

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

Definition at line 26 of file low\_saurion.h.

## 7.5.1.15 on\_wrote\_arg

```
void * on_wrote_arg
```

Additional argument for the write callback.

Definition at line 27 of file low\_saurion.h.

## 7.5.1.16 pool

ThreadPool\* pool

Thread pool for executing tasks in parallel.

Definition at line 9 of file low\_saurion.h.

## 7.5.1.17 rings

```
struct io_uring* rings
```

Array of io\_uring structures for managing the event queue.

Definition at line 0 of file low\_saurion.h.

## 7.5.1.18 ss

int ss

Server socket descriptor for accepting connections.

Definition at line 3 of file low\_saurion.h.

7.6 low\_saurion.h

#### 7.5.1.19 status

```
int status
```

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

Definition at line 8 of file low\_saurion.h.

## 7.5.1.20 status\_c

```
pthread_cond_t status_c
```

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

Definition at line 7 of file low saurion.h.

#### 7.5.1.21 status\_m

```
pthread_mutex_t status_m
```

Mutex to protect the state of the structure.

Definition at line 6 of file low\_saurion.h.

## 7.6 low\_saurion.h

```
00098 #ifndef LOW_SAURION_H
00099 #define LOW_SAURION_H
00100
00101 #include buring.h>
00102 #include <pthread.h>
00104 #include "config.h"
00105 #include "cthreadpool.hpp"
00106 #include "linked_list.h"
00107
00108 #ifdef __cplusplus
00109 extern "C" {
00110 #endif
00111
00129 #define PACKING_SZ 128
00130
00137 struct saurion {
00138 struct io_uring *rings;
00139 pthread_mutex_t
00140
              *m_rings;
00141
         int ss;
         int *efds;
struct Node *list;
00142
00143
         pthread_mutex_t status_m;
00145
         pthread_cond_t status_c;
00146
         int status;
00147
         ThreadPool *pool;
         uint32_t n_threads;
uint32_t next;
00148
00149
00159
         struct saurion_callbacks {
00166
            void (*on_connected)(const int fd, void *arg);
```

```
void *on_connected_arg;
          void (*on_readed) (const int fd, const void *const content, const ssize_t len, void *arg);
00178
          void *on_readed_arg;
00186
         void (*on_wrote) (const int fd, void *arg);
00187
         void *on_wrote_arg;
00195
         void (*on_closed) (const int fd, void *arg);
00196
         void *on_closed_arg;
00206
         void (*on_error)(const int fd, const char *const content, const ssize_t len, void *arg);
00207
         void *on_error_arg;
00208
           _attribute__((aligned(PACKING_SZ))) cb;
00209 } __attribute__((aligned(PACKING_SZ)));
00210
00214 int EXTERNAL_set_socket(int p);
00215
00228 [[nodiscard]]
00229 struct saurion *saurion_create(uint32_t n_threads);
00230
00242 [[nodiscard]]
00243 int saurion_start(struct saurion *s);
00244
00254 void saurion_stop(const struct saurion *s);
00255
00266 void saurion_destroy(struct saurion *s);
00267
00280 void saurion_send(struct saurion *s, const int fd, const char *const msq);
00282 #ifdef __cplusplus
00283
00284 #endif
00285
00286 #endif // !LOW_SAURION_H
00287
```

## 7.7 / 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)

## 7.8 low saurion secret.h

```
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 struct request {
```

```
00016
       void *prev;
00017
       size_t prev_size;
00018
       size_t prev_remain;
00019
       size_t next_iov;
00020
       size t next offset;
00021
       int event_type;
       size_t iovec_count;
00023
       int client_socket;
00024 struct iovec iov[];
00025 }
00059 [[nodiscard]]
00060 int allocate iovec(struct iovec *iov, size t amount, size t pos, size t size, void **chd ptr);
00095 int initialize_iovec(struct iovec *iov, size_t amount, size_t pos, const void *msg, size_t size,
00096
                          uint8_t h);
00097
00124 [[nodiscard]]
00125 int set_request (struct request **r, struct Node **l, size_t s, const void *m, uint8_t h);
00163 int read_chunk(void **dest, size_t *len, struct request *const req);
00164
00165 void free request (struct request *req, void **children ptr, size t amount);
00169 #ifdef __cplusplus
00170 }
00171 #endif
00172
00173 #endif // !LOW_SAURION_SECRET_H
```

## 7.9 /\_w/saurion/saurion/include/saurion.hpp File Reference

#include "low saurion.h"

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

#### **Classes**

class Saurion

## 7.10 saurion.hpp

```
00001 #ifndef SAURION HPP
00002 #define SAURION_HPP
00003
00004 #include "low_saurion.h"
00005
00006 class Saurion {
00007 public:
       using ConnectedCb = void (*)(const int, void *);
80000
00009
       using ReadedCb = void (*) (const int, const void *const, const ssize_t, void *);
       using WroteCb = void (*)(const int, void *);
00011
       using ClosedCb = void (*)(const int, void *);
00012
       using ErrorCb = void (*)(const int, const char *const, const ssize_t, void *);
00013
00014
       explicit Saurion(const uint32_t thds, const int sck) noexcept;
00015
       ~Saurion();
00016
00017
00018
       Saurion(Saurion &&) = delete;
       Saurion & operator = (const Saurion &) = delete;
00019
00020
       Saurion & operator = (Saurion & &) = delete;
00021
00022
       void init() noexcept;
00023
       void stop() noexcept;
00024
00025
       Saurion *on_connected(ConnectedCb ncb, void *arg) noexcept;
00026
       Saurion *on_readed(ReadedCb ncb, void *arg) noexcept;
00027
       Saurion *on_wrote(WroteCb ncb, void *arg) noexcept;
       Saurion *on_closed(ClosedCb ncb, void *arg) noexcept;
       Saurion *on_error(ErrorCb ncb, void *arg) noexcept;
```

```
00030
00031    void send(const int fd, const char *const msg) noexcept;
00032
00033    private:
00034    struct saurion *s;
00035    };
00036
00037 #endif // !SAURION_HPP
```

## 7.11 / w/saurion/saurion/include/threadpool.hpp File Reference

```
#include <bits/types/sig_atomic_t.h>
#include <pthread.h>
#include <cstdint>
#include <queue>
#include <unordered_map>
```

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

#### **Classes**

- struct Task
- class ThreadPool
- struct ThreadPool::AsyncQueue
- struct ThreadPool::AsyncMultiQueue

## **Enumerations**

enum StopFlags { KINDLY = 0x01 , FORCE = 0x02 }

## 7.11.1 Enumeration Type Documentation

## 7.11.1.1 StopFlags

```
enum StopFlags
```

## Enumerator



Definition at line 12 of file threadpool.hpp. 00012 { KINDLY = 0x01, FORCE = 0x02 };

## 7.12 threadpool.hpp

7.12 threadpool.hpp 77

```
00001 #ifndef THREADPOOL_HPP
00002 #define THREADPOOL_HPP
00003
00004 // Estructura para representar una tarea
00005 #include <bits/types/sig_atomic_t.h>
00006 #include <pthread.h>
00008 #include <cstdint>
00009 #include <queue>
00010 #include <unordered map>
00011
00012 enum StopFlags { KINDLY = 0x01, FORCE = 0x02 };
00013
00014 struct Task {
00015
       void (*function)(void*);
00016
       void* argument;
00017
00018
       explicit Task (void (*nfn) (void*), void* narg);
        Task(const Task&) = delete;
00019
00020
        Task(Task&&) = delete;
00021
        Task& operator=(const Task&) = delete;
00022
       Task& operator=(Task&&) = delete;
00023
       ~Task() = default;
00024 };
00025
00026 // Clase ThreadPool
00027 class ThreadPool {
00028 private:
00029
        typedef struct AsyncQueue {
00030
        private:
00031
         std::queue<Task*> m queue;
00032
          uint32_t m_max;
00033
          uint32_t m_cnt;
00034
          pthread_mutex_t m_mtx = PTHREAD_MUTEX_INITIALIZER;
00035
         public:
00036
00037
          explicit AsyncQueue(uint32 t cnt);
          ~AsyncQueue();
00039
          AsyncQueue(const AsyncQueue&) = delete;
00040
          AsyncQueue& operator=(const AsyncQueue&) = delete;
00041
          AsyncQueue(AsyncQueue&&) = delete;
00042
          AsyncQueue& operator=(AsyncQueue&&) = delete;
00043
00044
          void push(Task* task);
00045
          Task* front();
00046
          void pop();
00047
00048
          bool empty();
00049
        } AsyncQueue;
00050
        typedef struct AsyncMultiQueue {
         private:
00051
00052
         std::unordered_map<uint32_t, AsyncQueue*> m_queues;
00053
          std::unordered_map<uint32_t, AsyncQueue*>::iterator m_it;
00054
00055
         public:
00056
          explicit AsyncMultiQueue();
          ~AsyncMultiQueue();
00057
00058
          AsyncMultiQueue(const AsyncMultiQueue&) = delete;
00059
          AsyncMultiQueue& operator=(const AsyncMultiQueue&) = delete;
00060
          AsyncMultiQueue(AsyncMultiQueue&&) = delete;
00061
          AsyncMultiQueue& operator=(AsyncMultiQueue&&) = delete;
00062
00063
          void new_queue(uint32_t qid, uint32_t cnt);
00064
          void remove_queue(uint32_t qid);
00065
          void push(uint32_t qid, void (*nfn)(void*), void* arg);
Task* front(uint32_t& qid);
00066
00067
00068
          void pop(uint32_t qid);
00069
          void clear();
00070
00071
         bool empty();
00072
       } AsyncMultiQueue;
00073
      public:
00074
00075
        ThreadPool();
00076
        explicit ThreadPool(size_t num_threads);
00077
        ThreadPool(const ThreadPool&) = delete;
00078
        ThreadPool& operator=(const ThreadPool&) = delete;
00079
        ThreadPool(ThreadPool&&) = delete;
00080
        ThreadPool& operator=(ThreadPool&&) = delete;
00081
00082
        void init();
00083
        void stop();
00084
        void add(uint32_t qid, void (*nfn)(void*), void* arg);
       void add(void (*nfn)(void*), void* arg);
void new_queue(uint32_t qid, uint32_t cnt);
00085
00086
       void remove_queue (uint32_t qid);
00087
```

```
88000
       bool empty();
00089
00090 private:
00091
        void wait_closeable();
00092
00093 public:
       void wait_empty();
00095
        ~ThreadPool();
00096
00097 private:
       size_t m_nth;
00098
00099
        size t m started:
00100
        AsyncMultiQueue m_queues;
        pthread_mutex_t m_q_mtx = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t m_q_cond = PTHREAD_COND_INITIALIZER;
00101
00102
00103
        static pthread_mutex_t s_mtx;
00104
        pthread_t* m_ths;
        volatile sig_atomic_t m_fstop;
00105
00106
        volatile sig_atomic_t m_faccept;
00107
00108
        void thread_worker();
00109
        static void* thread_entry(void* arg);
00110 };
00111
00112 #endif
```

## 7.13 /\_w/saurion/saurion/src/cthreadpool.cpp File Reference

```
#include "cthreadpool.hpp"
Include dependency graph for cthreadpool.cpp:
```

#### **Functions**

- ThreadPool \* ThreadPool create (size t num threads)
- ThreadPool \* ThreadPool\_create\_default (void)
- void ThreadPool\_init (ThreadPool \*thp)
- void ThreadPool\_stop (ThreadPool \*thp)
- void ThreadPool\_add (ThreadPool \*thp, uint32\_t qid, void(\*nfn)(void \*), void \*arg)
- void ThreadPool add default (ThreadPool \*thp, void(\*nfn)(void \*), void \*arg)
- void ThreadPool\_new\_queue (ThreadPool \*thp, uint32\_t qid, uint32\_t cnt)
- void ThreadPool remove queue (ThreadPool \*thp, uint32 t qid)
- bool ThreadPool\_empty (ThreadPool \*thp)
- void ThreadPool\_wait\_empty (ThreadPool \*thp)
- void ThreadPool\_destroy (ThreadPool \*thp)

### 7.13.1 Function Documentation

#### 7.13.1.1 ThreadPool add()

## Definition at line 11 of file cthreadpool.cpp.

```
00011 { thp->add(qid, nfn, arg); }
```

### 7.13.1.2 ThreadPool\_add\_default()

Definition at line 13 of file cthreadpool.cpp.

```
00013 { thp->add(nfn, arg); }
```

## 7.13.1.3 ThreadPool\_create()

Definition at line 3 of file cthreadpool.cpp.

```
00003 { return new ThreadPool(num_threads); }
```

#### 7.13.1.4 ThreadPool create default()

Definition at line 5 of file cthreadpool.cpp.

```
00005 { return new ThreadPool(); }
```

## 7.13.1.5 ThreadPool\_destroy()

```
void ThreadPool_destroy ( {\tt ThreadPool} \ * \ thp \ )
```

Definition at line 23 of file cthreadpool.cpp.

```
00023 { delete thp; }
```

## 7.13.1.6 ThreadPool\_empty()

```
bool ThreadPool_empty ( {\tt ThreadPool} \ * \ thp \ )
```

Definition at line 19 of file cthreadpool.cpp.

```
00019 { return thp->empty(); }
```

## 7.13.1.7 ThreadPool\_init()

Definition at line 7 of file cthreadpool.cpp. 00007 { thp->init(); }

## 7.13.1.8 ThreadPool new queue()

Definition at line 15 of file cthreadpool.cpp. 00015 { thp->new\_queue(qid, cnt); }

#### 7.13.1.9 ThreadPool\_remove\_queue()

Definition at line 17 of file cthreadpool.cpp. 00017 { thp->remove\_queue(qid); }

## 7.13.1.10 ThreadPool\_stop()

```
void ThreadPool_stop (
          ThreadPool * thp )
```

Definition at line 9 of file cthreadpool.cpp. 00009 { thp->stop(); }

## 7.13.1.11 ThreadPool\_wait\_empty()

Definition at line 21 of file cthreadpool.cpp.

```
00021 { thp->wait_empty(); }
```

7.14 cthreadpool.cpp 81

## 7.14 cthreadpool.cpp

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

```
00001 #include "cthreadpool.hpp'
00002
00003 ThreadPool *ThreadPool_create(size_t num_threads) { return new ThreadPool(num_threads); }
00004
00005 ThreadPool *ThreadPool_create_default(void) { return new ThreadPool(); }
00006
00007 void ThreadPool_init(ThreadPool *thp) { thp->init(); }
80000
00009 void ThreadPool stop(ThreadPool *thp) { thp->stop(); }
00010
00011 void ThreadPool_add(ThreadPool *thp, uint32_t qid, void (*nfn) (void *), void *arg) { thp->add(qid,
     nfn, arg); }
00012
00013 void ThreadPool_add_default(ThreadPool *thp, void (*nfn) (void *), void *arg) { thp->add(nfn, arg); }
00014
00015 void ThreadPool_new_queue(ThreadPool *thp, uint32_t qid, uint32_t cnt) { thp->new_queue(qid, cnt); }
00016
00017 void ThreadPool_remove_queue(ThreadPool *thp, uint32_t qid) { thp->remove_queue(qid); }
00018
00019 bool ThreadPool_empty(ThreadPool *thp) { return thp->empty(); }
00020
00021 void ThreadPool_wait_empty(ThreadPool *thp) { thp->wait_empty(); }
00023 void ThreadPool_destroy(ThreadPool *thp) { delete thp; }
```

## 7.15 /\_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, void \*ptr)
- void list free (struct Node \*\*head)

## **Variables**

• pthread mutex t list mutex = PTHREAD MUTEX INITIALIZER

## 7.15.1 Function Documentation

#### 7.15.1.1 create\_node()

```
struct Node * create_node (
                  void * ptr,
                  size_t amount,
                  void ** children )
Definition at line 17 of file linked list.c.
         struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
// LCOV_EXCL_START
00018
00019
00020
         if (!new_node) {
00021
           return NULL;
00022
00023
         // LCOV_EXCL_STOP
         new_node->ptr = ptr;
new_node->size = amount;
00024
00025
00026
         new node->children = NULL;
00027
         if (amount > 0) {
00028
           new_node->children = (struct Node **) malloc(sizeof(struct Node *) * amount);
            // LCOV_EXCL_START
if (!new_node->children) {
00029
00030
00031
              free(new_node);
00032
              return NULL;
00033
00034
            // LCOV_EXCL_STOP
00035
            for (size_t i = 0; i < amount; ++i) {</pre>
00036
              new_node->children[i] = (struct Node *)malloc(sizeof(struct Node));
00037
              // LCOV_EXCL_START
00038
              if (!new_node->children[i]) {
   for (size_t j = 0; j < i; ++j) {</pre>
00039
00040
00041
                  free(new_node->children[j]);
00042
00043
                 free(new_node);
00044
                 return NULL;
00045
00046
00047
            // LCOV_EXCL_STOP
            for (size_t i = 0; i < amount; ++i) {
  new_node->children[i]->size = 0;
  new_node->children[i]->next = NULL;
00048
00049
00050
              new_node >children[i] ->ptr = children[i];
new_node ->children[i] ->children = NULL;
00051
00052
00053
            }
00054
         new_node->next = NULL;
00055
00056
         return new_node;
00057 }
```

## 7.15.1.2 free\_node()

free (current);

void free\_node (

00090

00091 }

#### 7.15.1.3 list\_delete\_node()

```
void list_delete_node (
               struct Node ** head,
               void * ptr)
Definition at line 93 of file linked_list.c.
00093
00094
        pthread_mutex_lock(&list_mutex);
00095
        struct Node *current = *head;
00096
        struct Node *prev = NULL;
00097
00098
        // Si el nodo a eliminar es el nodo cabeza
00099
        if (current && current->ptr == ptr) {
        *head = current->next;
free_node(current);
00100
00101
00102
        pthread_mutex_unlock(&list_mutex);
00103
          return;
00104 }
00105
        // Buscar el nodo a eliminar
00106
00107
        while (current && current->ptr != ptr) {
        prev = current;
00108
00109
          current = current->next;
00110
00111
       // Si el dato no se encuentra en la lista
if (!current) {
00112
00113
        pthread_mutex_unlock(&list_mutex);
00114
00115
         return;
00116
00117
00118
       // Desenlazar el nodo y liberarlo
00119
       prev->next = current->next;
00120
        free_node(current);
00121
       pthread_mutex_unlock(&list_mutex);
00122 }
```

## 7.15.1.4 list\_free()

void list\_free (

```
Definition at line 124 of file linked list.c.
00124
00125
        pthread_mutex_lock(&list_mutex);
00126
        struct Node *current = *head;
00127
       struct Node *next;
00128
00129
       while (current) {
        next = current->next;
free_node(current);
00130
00131
          current = next;
00133
00134
00135
        *head = NULL;
```

00136 pthread\_mutex\_unlock(&list\_mutex);
00137 }

struct Node \*\* head )

#### 7.15.1.5 list\_insert()

```
size_t amount,
void ** children )
```

#### Definition at line 59 of file linked\_list.c.

```
00059
        struct Node *new_node = create_node(ptr, amount, children);
// LCOV_EXCL_START
00061
00062
        if (!new_node) {
   return 1;
00063
00064
        // LCOV_EXCL_STOP
00065
00066
        pthread_mutex_lock(&list_mutex);
00067
        if (!*head) {
  *head = new_node;
00068
00069
         pthread_mutex_unlock(&list_mutex);
00070
          return 0;
00071
00072
        struct Node *temp = *head;
00073
        while (temp->next) {
00074
          temp = temp->next;
00076
        temp->next = new_node;
00077
        pthread_mutex_unlock(&list_mutex);
00078
         return 0;
00079 }
```

### 7.15.2 Variable Documentation

#### 7.15.2.1 list mutex

```
pthread_mutex_t list_mutex = PTHREAD_MUTEX_INITIALIZER
```

Definition at line 14 of file linked list.c.

## 7.16 linked\_list.c

```
00001 #include "linked_list.h"
00002
00003 #include <pthread.h>
00004 #include <stdlib.h>
00005
00006 struct Node {
00007 void *ptr;
00008 size_t size;
00009 struct Node **children;
00010
        struct Node *next;
00012
00013 // Global mutex for thread safety
00014 pthread_mutex_t list_mutex = PTHREAD_MUTEX_INITIALIZER;
00015
00016 // Función para crear un nuevo nodo
00017 struct Node *create_node(void *ptr, size_t amount, void **children) {
00018 struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
        // LCOV_EXCL_START
if (!new_node) {
00019
00020
00021
           return NULL;
00022
        // LCOV_EXCL_STOP
00024
        new_node->ptr = ptr;
00025
         new_node->size = amount;
00026
         new_node->children = NULL;
00027
         if (amount > 0) {
         new_node->children = (struct Node **)malloc(sizeof(struct Node *) * amount);
00028
00029
           // LCOV_EXCL_START
           if (!new_node->children) {
```

7.16 linked list.c 85

```
00031
             free (new_node);
            return NULL;
00032
00033
           // LCOV_EXCL_STOP
00034
          for (size_t i = 0; i < amount; ++i) {
  new_node->children[i] = (struct Node *)malloc(sizeof(struct Node));
00035
00036
00038
             // LCOV_EXCL_START
00039
             if (!new_node->children[i]) {
               for (size_t j = 0; j < i; ++j) {
  free(new_node->children[j]);
00040
00041
00042
00043
               free (new_node);
              return NULL;
00044
00045
00046
           // LCOV_EXCL_STOP
00047
00048
          for (size_t i = 0; i < amount; ++i) {
  new_node->children[i]->size = 0;
00050
             new_node->children[i]->next = NULL;
00051
             new_node->children[i]->ptr = children[i];
00052
            new_node->children[i]->children = NULL;
00053
          }
00054
00055
        new_node->next = NULL;
00056
        return new_node;
00057 }
00058
00059 int list_insert(struct Node **head, void *ptr, size_t amount, void **children) {
        struct Node *new_node = create_node(ptr, amount, children);
// LCOV_EXCL_START
00060
00061
00062
        if (!new_node) {
00063
          return 1;
00064
00065
        // LCOV_EXCL_STOP
00066
        pthread_mutex_lock(&list_mutex);
00067
        if (!*head) {
00068
          *head = new_node;
00069
          pthread_mutex_unlock(&list_mutex);
00070
          return 0;
00071
00072
        struct Node *temp = *head;
00073
        while (temp->next) {
00074
          temp = temp->next;
00075
00076
        temp->next = new_node;
00077
        pthread_mutex_unlock(&list_mutex);
00078
        return 0;
00079 }
08000
00081 void free_node(struct Node *current) {
00082
        if (current->size > 0) {
00083
          for (size_t i = 0; i < current->size; ++i) {
            free(current->children[i]->ptr);
00084
00085
             free(current->children[i]);
00086
          free(current->children);
00088
00089
        free(current->ptr);
00090
        free(current);
00091 }
00092
00093 void list_delete_node(struct Node **head, void *ptr) {
00094
      pthread_mutex_lock(&list_mutex);
00095
        struct Node *current = *head;
00096
        struct Node *prev = NULL;
00097
00098
        // Si el nodo a eliminar es el nodo cabeza
00099
        if (current && current->ptr == ptr) {
00100
         *head = current->next;
00101
           free_node(current);
00102
          pthread_mutex_unlock(&list_mutex);
00103
          return;
00104
00105
00106
        // Buscar el nodo a eliminar
00107
        while (current && current->ptr != ptr) {
00108
        prev = current;
00109
          current = current->next;
00110
00111
00112
        // Si el dato no se encuentra en la lista
00113
        if (!current) {
00114
          pthread_mutex_unlock(&list_mutex);
           return;
00115
00116
00117
```

```
// Desenlazar el nodo y liberarlo
00119 prev->next = current->next;
00120 free_node(current);
        free_node(current);
00121 pthread_mutex_unlock(&list_mutex);
00122 }
00123
00124 void list_free(struct Node **head) {
00125 pthread_mutex_lock(&list_mutex);
00126
        struct Node *current = *head;
00127
        struct Node *next;
00128
00129
        while (current) {
        next = current->next;
free_node(current);
00130
00131
00132
          current = next;
00133
00134
        *head = NULL;
00135
       pthread_mutex_unlock(&list_mutex);
00136
00137 }
```

## 7.17 /\_w/saurion/saurion/src/low\_saurion.c File Reference

```
#include "low_saurion.h"
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/eventfd.h>
Include dependency graph for low_saurion.c:
```

## **Classes**

- · struct request
- · struct saurion\_wrapper

#### **Macros**

```
    #define EVENT TYPE ACCEPT 0

• #define EVENT_TYPE_READ 1
```

- #define EVENT TYPE WRITE 2
- #define EVENT\_TYPE\_WAIT 3
- #define EVENT TYPE ERROR 4
- #define MIN(a, b) ((a) < (b) ? (a) : (b))
- #define MAX(a, b) ((a) > (b) ? (a) : (b))

### **Functions**

- static uint32\_t next (struct saurion \*s)
- static uint64\_t htonll (uint64\_t value)
- static uint64 t ntohll (uint64 t value)
- void free\_request (struct request \*req, void \*\*children\_ptr, size\_t amount)
- int initialize\_iovec (struct iovec \*iov, size\_t amount, size\_t pos, const void \*msg, size\_t size, uint8\_t h) Initializes a specified iovec structure with a message fragment.
- int allocate iovec (struct iovec \*iov, size t amount, size t pos, size t size, void \*\*chd ptr)
- int set\_request (struct request \*\*r, struct Node \*\*I, size\_t s, const void \*m, uint8\_t h)

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

- static void add\_accept (struct saurion \*const s, const struct sockaddr\_in \*const ca, socklen\_t \*const cal)
- 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 reg)
- 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 EXTERNAL\_set\_socket (const int p)
- struct saurion \* saurion\_create (uint32\_t n\_threads)

Creates an instance of the saurion structure.

- void saurion\_worker\_master (void \*arg)
- 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

## 7.17.1 Macro Definition Documentation

#### 7.17.1.1 EVENT\_TYPE\_ACCEPT

```
#define EVENT_TYPE_ACCEPT 0
```

Definition at line 9 of file low\_saurion.c.

## 7.17.1.2 EVENT\_TYPE\_ERROR

```
#define EVENT_TYPE_ERROR 4
```

Definition at line 13 of file low saurion.c.

## 7.17.1.3 EVENT\_TYPE\_READ

```
#define EVENT_TYPE_READ 1
```

Definition at line 10 of file low\_saurion.c.

## 7.17.1.4 EVENT\_TYPE\_WAIT

```
#define EVENT_TYPE_WAIT 3
```

Definition at line 12 of file low\_saurion.c.

## 7.17.1.5 EVENT\_TYPE\_WRITE

```
#define EVENT_TYPE_WRITE 2
```

Definition at line 11 of file low\_saurion.c.

## 7.17.1.6 MAX

Definition at line 28 of file low\_saurion.c.

## 7.17.1.7 MIN

Definition at line 27 of file low\_saurion.c.

## 7.17.2 Function Documentation

### 7.17.2.1 add\_accept()

```
static void add_accept (
               struct saurion *const s.
                const struct sockaddr_in *const ca,
                socklen_t *const cal ) [static]
Definition at line 186 of file low saurion.c.
00188
         int res = ERROR_CODE;
00189
        pthread_mutex_lock(&s->m_rings[0]);
00190
         while (res != SUCCESS_CODE) {
00191
          struct io_uring_sqe *sqe = io_uring_get_sqe(&s->rings[0]);
00192
          while (!sqe) {
00193
            sqe = io_uring_get_sqe(&s->rings[0]);
00194
            usleep(TIMEOUT_RETRY);
00195
00196
          struct request *req = NULL;
00197
          if (!set_request(&req, &s->list, 0, NULL, 0)) {
00198
            free (sge);
            usleep(TIMEOUT_RETRY);
res = ERROR_CODE;
00199
00200
00201
            continue;
00202
          req->client_socket = 0;
req->event_type = EVENT_TYPE_ACCEPT;
00203
00204
          io_uring_prep_accept(sqe, s->ss, (struct sockaddr *)ca, cal, 0);
io_uring_sqe_set_data(sqe, req);
00205
00206
00207
          if (io_uring_submit(&s->rings[0]) < 0) {</pre>
00208
            free(sqe);
00209
            list_delete_node(&s->list, req);
            usleep(TIMEOUT_RETRY);
00210
00211
            res = ERROR_CODE;
00212
            continue;
00213
00214
          res = SUCCESS_CODE;
00215
00216
        pthread_mutex_unlock(&s->m_rings[0]);
00217 }
```

## 7.17.2.2 add\_efd()

#### Definition at line 219 of file low\_saurion.c.

```
00220
        pthread_mutex_lock(&s->m_rings[sel]);
00221
        int res = ERROR_CODE;
        while (res != SUCCESS_CODE) {
00222
00223
         struct io_uring *ring = &s->rings[sel];
          struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
00225
          while (!sqe) {
00226
            sqe = io_uring_get_sqe(ring);
00227
            usleep(TIMEOUT_RETRY);
00228
00229
          struct request *req = NULL;
          if (!set_request(&req, &s->list, CHUNK_SZ, NULL, 0)) {
00230
00231
             free(sqe);
00232
             res = ERROR_CODE;
00233
            continue;
00234
          req->event_type = EVENT_TYPE_READ;
00235
           req->client_socket = client_socket;
00236
00237
           io_uring_prep_readv(sqe, client_socket, &req->iov[0], req->iovec_count, 0);
          io_uring_sqe_set_data(sqe, req);
if (io_uring_submit(ring) < 0) {</pre>
00238
00239
00240
            free(sqe);
            list_delete_node(&s->list, req);
res = ERROR_CODE;
00241
00242
            continue;
```

```
00244     }
00245     res = SUCCESS_CODE;
00246     }
00247     pthread_mutex_unlock(&s->m_rings[sel]);
00248 }
```

#### 7.17.2.3 add read()

#### Definition at line 250 of file low saurion.c.

```
{
00251
         int sel = next(s);
00252
         int res = ERROR_CODE;
00253
         pthread_mutex_lock(&s->m_rings[sel]);
00254
         while (res != SUCCESS_CODE) {
          struct io_uring *ring = &s->rings[sel];
00255
00256
          struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
00257
          while (!sqe) {
00258
           sqe = io_uring_get_sqe(ring);
00259
             usleep(TIMEOUT_RETRY);
00260
00261
          struct request *req = NULL;
          if (!set_request(&req, &s->list, CHUNK_SZ, NULL, 0)) {
00262
00263
           free(sqe);
00264
            res = ERROR_CODE;
00265
            continue;
00266
          req->event_type = EVENT_TYPE_READ;
req->client_socket = client_socket;
00267
00268
00269
           io_uring_prep_readv(sqe, client_socket, &req->iov[0], req->iovec_count, 0);
00270
           io_uring_sqe_set_data(sqe, req);
00271
           if (io_uring_submit(ring) < 0) {</pre>
00272
           free(sqe);
            list_delete_node(&s->list, req);
res = ERROR_CODE;
00273
00274
00275
             continue;
00276
00277
          res = SUCCESS_CODE;
.
00279 pthread_mutex_unlock(&s->m_rings[sel]);
00280 }
```

### 7.17.2.4 add\_read\_continue()

```
static void add_read_continue (
    struct saurion *const s,
    struct request * oreq,
    const int sel ) [static]
```

#### Definition at line 282 of file low saurion.c.

```
00282
00283
        pthread_mutex_lock(&s->m_rings[sel]);
00284
        int res = ERROR_CODE;
00285
        while (res != SUCCESS_CODE) {
00286
         struct io_uring *ring = &s->rings[sel];
00287
          struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
00288
         while (!sqe) {
00289
           sqe = io_uring_get_sqe(ring);
00290
            usleep(TIMEOUT_RETRY);
00291
          if (!set_request(&oreq, &s->list, oreq->prev_remain, NULL, 0)) {
00292
00293
            free(sqe);
00294
            res = ERROR_CODE;
00295
            continue;
00296
```

{

```
00297
            io_uring_prep_readv(sqe, oreq->client_socket, &oreq->iov[0], oreq->iovec_count, 0);
           io_uring_sqe_set_data(sqe, oreq);
if (io_uring_submit(ring) < 0) {</pre>
00298
00299
00300
             free(sqe);
             list_delete_node(&s->list, oreq);
res = ERROR_CODE;
00301
00302
00303
             continue;
00304
00305
           res = SUCCESS_CODE;
00306
        pthread_mutex_unlock(&s->m_rings[sel]);
00307
00308 }
```

### 7.17.2.5 add\_write()

#### Definition at line 310 of file low saurion.c.

```
00310
        int res = ERROR_CODE;
00311
        pthread_mutex_lock(&s->m_rings[sel]);
while (res != SUCCESS_CODE) {
00312
00313
00314
          struct io_uring *ring = &s->rings[sel];
00315
          struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
00316
          while (!sqe) {
           sqe = io_uring_get_sqe(ring);
00317
            usleep(TIMEOUT_RETRY);
00318
00319
00320
          struct request *req = NULL;
00321
          if (!set_request(&req, &s->list, strlen(str), (void *)str, 1)) {
           free(sqe);
00322
00323
            res = ERROR CODE;
00324
            continue;
00325
00326
          req->event_type = EVENT_TYPE_WRITE;
00327
          req->client_socket = fd;
00328
           io_uring_prep_writev(sqe, req->client_socket, req->iov, req->iovec_count, 0);
          io_uring_sqe_set_data(sqe, req);
if (io_uring_submit(ring) < 0) {</pre>
00329
00330
           free(sqe);
list_delete_node(&s->list, req);
00331
00332
00333
            res = ERROR_CODE;
00334
            usleep(TIMEOUT_RETRY);
00335
             continue;
00336
          res = SUCCESS_CODE;
00337
00338
        pthread_mutex_unlock(&s->m_rings[sel]);
00339
00340 }
```

## 7.17.2.6 handle accept()

#### 7.17.2.7 handle\_close()

```
static void handle_close (
               const struct saurion *const s,
              const struct request *const req ) [static]
Definition at line 540 of file low_saurion.c.
00540
00541
       s->cb.on_closed(req->client_socket, s->cb.on_closed_arg);
}
        if (s->cb.on closed) {
00542
00543
00544
       close(req->client_socket);
00545 }
7.17.2.8 handle_error()
static void handle_error (
              const struct saurion *const s,
              const struct request *const req ) [static]
Definition at line 533 of file low saurion.c.
00534
        if (s->cb.on_error) {
         const char *resp = "ERROR";
00535
00536
          s->cb.on_error(req->client_socket, resp, (ssize_t)strlen(resp), s->cb.on_error_arg);
00537
00538 }
7.17.2.9 handle_read()
static void handle_read (
              struct saurion *const s,
              struct request *const req ) [static]
Definition at line 495 of file low saurion.c.
00495
00496
        void *msg = NULL;
        size_t len = 0;
while (1) {
00497
00498
00499
         if (!read_chunk(&msg, &len, req)) {
00500
00501
         // Hay siguiente
00502
00503
         if (req->next_iov || req->next_offset) {
           if (s->cb.on_readed && msg) {
00505
             s->cb.on_readed(req->client_socket, msg, len, s->cb.on_readed_arg);
00506
00507
           free(msg);
00508
           msg = NULL;
continue;
00509
00510
00511
          // Hay previo pero no se ha completado
00512
          if (req->prev && req->prev_size && req->prev_remain) {
           add_read_continue(s, req, next(s));
00513
00514
            return;
00515
00516
          // Hay un único mensaje y se ha completado
00517
          if (s->cb.on_readed && msg) {
00518
           s->cb.on_readed(req->client_socket, msg, len, s->cb.on_readed_arg);
00519
00520
          free (msg):
00521
          msg = NULL;
00522
         break;
00523
00524
        add_read(s, req->client_socket);
00525 }
```

#### 7.17.2.10 handle\_write()

```
static void handle_write (
            const struct saurion *const s,
             const int fd ) [static]
Definition at line 527 of file low saurion.c.
00528
       if (s->cb.on_wrote) {
00529
        s->cb.on_wrote(fd, s->cb.on_wrote_arg);
00530 }
00531 }
```

#### 7.17.2.11 htonll()

static uint64\_t htonll (

```
uint64_t value ) [static]
Definition at line 41 of file low_saurion.c.
00042
              int num = 42;
            if (*(char *)&num == 42) { // Little endian check
  uint32_t high_part = htonl((uint32_t) (value » 32));
  uint32_t low_part = htonl((uint32_t) (value & 0xFFFFFFFLL));
  return ((uint64_t)low_part « 32) | high_part;
00043
00044
00045
00046
00047
00048 return value;
00049 }
```

#### 7.17.2.12 next()

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

#### Definition at line 36 of file low saurion.c.

```
00036
00037
          s\rightarrow next = (s\rightarrow next + 1) % s\rightarrow n_threads;
00038
          return s->next;
00039 }
```

#### 7.17.2.13 ntohll()

```
static uint64_t ntohll (
                    uint64_t value ) [static]
Definition at line 51 of file low saurion.c.
00052
00053
          if (*(char *)&num == 42) { // Little endian check
           uint32_t high_part = ntohl((uint32_t) (value » 32));
uint32_t low_part = ntohl((uint32_t) (value & 0xFFFFFFFLL));
return ((uint64_t)low_part « 32) | high_part;
```

00057 } 00058 return value;

00054 00055 00056

00059 }

#### 7.17.2.14 saurion\_worker\_master()

```
void saurion_worker_master (
               void * arg )
Definition at line 677 of file low saurion.c.
00677
00678
        struct saurion *const s = (struct saurion *)arg;
        struct io_uring ring = s->rings[0];
00680
       struct io_uring_cqe *cqe = NULL;
00681
        struct sockaddr_in client_addr;
00682
       socklen_t client_addr_len = sizeof(client_addr);
00683
00684
        add efd(s, s->efds[0], 0);
       add_accept(s, &client_addr, &client_addr_len);
00685
00686
00687
        pthread_mutex_lock(&s->status_m);
00688
        s->status = 1;
        pthread_cond_signal(&s->status_c);
00689
00690
        pthread_mutex_unlock(&s->status_m);
00691
        while (1) {
         int ret = io_uring_wait_cqe(&ring, &cqe);
if (ret < 0) {</pre>
00692
00693
00694
            free(cqe);
00695
            return;
00696
          struct request *req = (struct request *)cqe->user_data;
00697
00698
00699
           io_uring_cqe_seen(&s->rings[0], cqe);
00700
00701
00702
          if (cqe->res < 0) {
00703
            list_delete_node(&s->list, req);
00704
            return;
00705
00706
          if (req->client_socket == s->efds[0]) {
00707
            io_uring_cqe_seen(&s->rings[0], cqe);
00708
            list_delete_node(&s->list, req);
00709
            break:
00710
00711
          /* Mark this request as processed */
00712
          io_uring_cqe_seen(&s->rings[0], cqe);
00713
          switch (req->event_type) {
00714
            case EVENT_TYPE_ACCEPT:
            handle_accept(s, cqe->res);
00715
00716
             add_accept(s, &client_addr, &client_addr_len);
00717
             add_read(s, cqe->res);
00718
             list_delete_node(&s->list, req);
            break;
case EVENT_TYPE_READ:
00719
00720
00721
             if (cqe->res < 0) {
00722
               handle_error(s, req);
00723
00724
              if (cqe->res < 1) {
00725
               handle_close(s, req);
00726
00727
              if (cge->res > 0) {
                handle_read(s, req);
00728
00729
00730
              list_delete_node(&s->list, req);
00731
            case EVENT_TYPE_WRITE:
00732
             handle_write(s, req->client_socket);
list_delete_node(&s->list, req);
00733
00734
              break;
00736
         }
00737
00738
       pthread_mutex_lock(&s->status_m);
00739
        s->status = 2;
00740
       pthread_cond_signal(&s->status_c);
00741
        pthread_mutex_unlock(&s->status_m);
00742
        return;
00743 }
```

## 7.17.2.15 saurion\_worker\_slave()

```
Definition at line 745 of file low_saurion.c.
        struct saurion_wrapper *const ss = (struct saurion_wrapper *)arg;
00746
        struct saurion *s = ss->s;
const int sel = ss->sel;
00747
00748
00749
        free(ss);
00750
       struct io_uring ring = s->rings[sel];
00751
        struct io_uring_cqe *cqe = NULL;
00752
00753
       add_efd(s, s->efds[sel], sel);
00754
00755
       pthread_mutex_lock(&s->status m);
00756
        s->status = 1;
00757
        pthread_cond_signal(&s->status_c);
00758
        pthread_mutex_unlock(&s->status_m);
00759
        while (1) {
00760
         int ret = io_uring_wait_cqe(&ring, &cqe);
00761
         if (ret < 0) {
00762
           free(cqe);
00763
            return;
00764
00765
          struct request *req = (struct request *)cqe->user_data;
00766
          if (!req) {
00767
            io_uring_cqe_seen(&ring, cqe);
00768
            continue;
00769
00770
          if (cqe->res < 0) {</pre>
00771
           list_delete_node(&s->list, req);
00772
            return;
00773
00774
          if (req->client_socket == s->efds[sel]) {
00775
           io_uring_cqe_seen(&ring, cqe);
00776
            list_delete_node(&s->list, req);
00777
            break;
00778
00779
          ^{\prime} /* Mark this request as processed */
         00780
00781
00782
           case EVENT_TYPE_READ:
00783
             if (cqe->res < 0) {
00784
                handle_error(s, req);
00785
             if (cqe->res < 1) {
00786
00787
               handle_close(s, req);
00788
00789
              if (cqe->res > 0) {
00790
               handle_read(s, req);
00791
00792
              list_delete_node(&s->list, reg);
00793
              break:
00794
            case EVENT_TYPE_WRITE:
00795
             handle_write(s, req->client_socket);
00796
              list_delete_node(&s->list, req);
00797
              break;
00798
         }
00799
00800
       pthread_mutex_lock(&s->status_m);
00801
        s->status = 2;
00802
        pthread_cond_signal(&s->status_c);
00803
        pthread_mutex_unlock(&s->status_m);
00804
        return;
00805 }
```

#### 7.17.3 Variable Documentation

## 7.17.3.1 print\_mutex

pthread\_mutex\_t print\_mutex

Definition at line 29 of file low\_saurion.c.

## 7.18 low saurion.c

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

```
00001 #include "low_saurion.h"
00002
00003 #include <netinet/in.h>
00004 #include <stdio.h>
00005 #include <stdlib.h>
00006 #include <string.h>
00007 #include <sys/eventfd.h>
80000
00009 #define EVENT_TYPE_ACCEPT 0
00010 #define EVENT_TYPE_READ 1
00011 #define EVENT_TYPE_WRITE 2
00012 #define EVENT_TYPE_WAIT 3
00013 #define EVENT_TYPE_ERROR 4
00014
00015 struct request {
00016
       void *prev;
00017
       size_t prev_size;
00018
       size_t prev_remain;
00019
       size_t next_iov;
00020
        size_t next_offset;
00021
        int event_type;
       size_t iovec_count;
00023
        int client_socket;
00024
      struct iovec iov[];
00025 };
00026
00027 #define MIN(a, b) ((a) < (b) ? (a) : (b)) 00028 #define MAX(a, b) ((a) > (b) ? (a) : (b))
00029 pthread_mutex_t print_mutex;
00031 struct saurion_wrapper {
00032 struct saurion *s;
00033 uint32_t sel;
00034 };
00035
00036 static uint32_t next(struct saurion *s) {
00037 s->next = (s->next + 1) % s->n_threads;
       return s->next;
00038
00039 }
00040
00041 static uint64_t htonl1(uint64_t value) {
00042 int num = 42;
00043
        if (*(char *)&num == 42) { // Little endian check
        uint32_t high_part = htonl((uint32_t) (value » 32));
uint32_t low_part = htonl((uint32_t) (value & 0xFFFFFFFLL));
00044
00045
00046
          return ((uint64_t)low_part « 32) | high_part;
00047
        return value;
00049 }
00050
00051 static uint64_t ntohl1(uint64_t value) {
00052
        int num = 42;
        if (*(char *)&num == 42) { // Little endian check
00053
        uint32_t high_part = ntohl((uint32_t)(value & 0xfffffffLL));
uint32_t low_part = ntohl((uint32_t)(value & 0xffffffffLL));
00054
00055
00056
          return ((uint64_t)low_part « 32) | high_part;
00057
00058
        return value:
00059 }
00060
00061 void free_request(struct request *req, void **children_ptr, size_t amount) {
00062 if (children_ptr) {
00063
          free(children_ptr);
00064
          children_ptr = NULL;
00065
00066
        for (size_t i = 0; i < amount; ++i) {</pre>
        free(req->iov[i].iov_base);
00067
00068
          req->iov[i].iov_base = NULL;
00069
00070
        free (req);
00071
        req = NULL;
        free(children_ptr);
00072
        children_ptr = NULL;
00074 }
00075
00076 [[nodiscard]]
00077 int initialize_iovec(struct iovec *iov, size_t amount, size_t pos, const void *msg, size_t size,
00078
                            uint8 t h) {
        if (!iov || !iov->iov_base) {
00082
```

7.18 low\_saurion.c 97

```
size_t len = iov->iov_len;
          char *dest = (char *)iov->iov_base;
char *orig = (char *)msg + pos * CHUNK_SZ;
00084
00085
00086
          size_t cpy_sz = 0;
00087
          if (h) {
00088
            if (pos == 0) {
             uint64_t send_size = htonl1(size);
00090
              memcpy(dest, &send_size, sizeof(uint64_t));
               dest += sizeof(uint64_t);
00091
00092
              len -= sizeof(uint64_t);
            } else {
00093
              orig -= sizeof(uint64_t);
00094
00095
00096
            if ((pos + 1) == amount) {
00097
00098
               cpy_sz = (len < size ? len : size);</pre>
00099
               dest[cpy_sz] = 0;
            }
00100
00101
00102
          cpy_sz = (len < size ? len : size);</pre>
00103
          memcpy(dest, orig, cpy_sz);
          dest += cpy_sz;
00104
          size_t rem = CHUNK_SZ - (dest - (char *)iov->iov_base);
00105
00106
          memset(dest, 0, rem);
00107
        } else {
         memset((char *)iov->iov_base, 0, CHUNK_SZ);
00109
00110
        return SUCCESS_CODE;
00111 }
00112
00113 [[nodiscard]]
00114 int allocate_iovec(struct iovec *iov, size_t amount, size_t pos, size_t size, void **chd_ptr) {
00115    if (!iov || !chd_ptr) {
00116
          return ERROR_CODE;
00117
        iov->iov_base = malloc(CHUNK_SZ);
00118
        if (!iov->iov_base) {
00119
          return ERROR_CODE;
00121
        iov->iov_len = (pos == (amount - 1) ? (size % CHUNK_SZ) : CHUNK_SZ);
if (iov->iov_len == 0) {
00122
00123
         iov->iov_len = CHUNK_SZ;
00124
00125
00126
        chd_ptr[pos] = iov->iov_base;
00127
        return SUCCESS_CODE;
00128 }
00129
00130 [[nodiscard]]
00131 int set_request (struct request **r, struct Node **l, size_t s, const void *m, uint8_t h) {
if (h) {
00134
          full_size += (sizeof(uint64_t) + 1);
00135
00136
        size t amount = full size / CHUNK SZ;
        amount = amount + (full_size % CHUNK_SZ == 0 ? 0 : 1);
00137
        struct request *temp =
00138
            (struct request *)malloc(sizeof(struct request) + sizeof(struct iovec) * amount);
00140
        if (!temp) {
00141
         return ERROR_CODE;
00142
        if (!*r) {
00143
00144
         *r = temp;
00145
          (*r)->prev = NULL;
00146
           (*r) \rightarrow prev\_size = 0;
00147
           (*r) ->prev_remain = 0;
00148
           (*r) ->next_iov = 0;
00149
           (*r) \rightarrow next_offset = 0;
00150
        } else {
00151
          temp->client_socket = (*r)->client_socket;
00152
          temp->event_type = (*r)->event_type;
00153
          temp->prev = (*r)->prev;
          temp->prev_size = (*r)->prev_size;
temp->prev_remain = (*r)->prev_remain;
00154
00155
          temp->next_iov = (*r)->next_iov;
00156
          temp->next_offset = (*r)->next_offset;
00157
00158
          *r = temp;
00159
00160
        struct request *req = *r;
        req->iovec_count = (int)amount;
00161
        void **children_ptr = (void **)malloc(amount * sizeof(void *));
00162
        if (!children_ptr) {
  free_request(req, children_ptr, 0);
00163
00164
          return ERROR_CODE;
00165
00166
00167
        for (size_t i = 0; i < amount; ++i) {</pre>
          if (!allocate_iovec(&req->iov[i], amount, i, full_size, children_ptr)) {
00168
00169
            free_request(req, children_ptr, amount);
```

```
return ERROR_CODE;
00171
00172
          if (!initialize_iovec(&req->iov[i], amount, i, m, s, h)) {
00173
           free_request(req, children_ptr, amount);
00174
            return ERROR_CODE;
00175
00176
00177
        if (list_insert(l, req, amount, children_ptr)) {
        free_request(req, children_ptr, amount);
00178
00179
         return ERROR_CODE;
00180
00181
       free (children_ptr);
00182
       return SUCCESS_CODE;
00183 }
00184
00186 static void add_accept(struct saurion *const s, const struct sockaddr_in *const ca,
00187
                            socklen_t *const cal) {
       int res = ERROR_CODE;
00188
00189
       pthread_mutex_lock(&s->m_rings[0]);
00190
        while (res != SUCCESS_CODE) {
00191
         struct io_uring_sqe *sqe = io_uring_get_sqe(&s->rings[0]);
00192
         while (!sqe) {
00193
           sqe = io_uring_get_sqe(&s->rings[0]);
00194
           usleep (TIMEOUT_RETRY);
00195
00196
         struct request *req = NULL;
00197
         if (!set_request(&req, &s->list, 0, NULL, 0)) {
00198
           free(sqe);
            usleep (TIMEOUT_RETRY);
00199
00200
           res = ERROR CODE:
00201
           continue;
00202
00203
          req->client_socket = 0;
          req->event_type = EVENT_TYPE_ACCEPT;
00204
00205
          io_uring_prep_accept(sqe, s->ss, (struct sockaddr \star)ca, cal, 0);
00206
          io_uring_sqe_set_data(sqe, req);
          if (io_uring_submit(&s->rings[0]) < 0) {</pre>
00208
           free(sqe);
00209
            list_delete_node(&s->list, req);
00210
            usleep(TIMEOUT_RETRY);
00211
           res = ERROR CODE;
00212
           continue;
00213
00214
         res = SUCCESS_CODE;
00215
00216 pthread_mutex_unlock(&s->m_rings[0]);
00217 }
00218
00219 static void add_efd(struct saurion *const s, const int client_socket, int sel) {
       pthread_mutex_lock(&s->m_rings[sel]);
00221
        int res = ERROR_CODE;
00222
        while (res != SUCCESS_CODE) {
00223
         struct io_uring *ring = &s->rings[sel];
00224
         struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
00225
         while (!sge) {
          sqe = io_uring_get_sqe(ring);
00227
           usleep(TIMEOUT_RETRY);
00228
00229
          struct request *req = NULL;
         if (!set_request(&req, &s->list, CHUNK_SZ, NULL, 0)) {
00230
00231
          free(sge);
00232
           res = ERROR_CODE;
00233
           continue;
00234
00235
          req->event_type = EVENT_TYPE_READ;
00236
          req->client_socket = client_socket;
io_uring_prep_readv(sqe, client_socket, &req->iov[0], req->iovec_count, 0);
00237
00238
          io_uring_sqe_set_data(sqe, req);
            (io_uring_submit(ring) < 0) {
00239
00240
          free(sqe);
00241
            list_delete_node(&s->list, req);
00242
           res = ERROR_CODE;
00243
           continue;
00244
00245
         res = SUCCESS CODE;
00246
00247
        pthread_mutex_unlock(&s->m_rings[sel]);
00248 }
00249
00250 static void add_read(struct saurion *const s, const int client_socket) {
       int sel = next(s);
00252
        int res = ERROR_CODE;
00253
        pthread_mutex_lock(&s->m_rings[sel]);
00254
       while (res != SUCCESS_CODE) {
         struct io_uring *ring = &s->rings[sel];
struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
00255
00256
```

```
while (!sqe) {
00258
           sqe = io_uring_get_sqe(ring);
00259
            usleep(TIMEOUT_RETRY);
00260
00261
          struct request *req = NULL;
          if (!set_request(&req, &s->list, CHUNK_SZ, NULL, 0)) {
00262
00263
           free(sqe);
00264
            res = ERROR_CODE;
00265
            continue;
00266
00267
          req->event_type = EVENT_TYPE_READ;
          req->client_socket = client_socket;
00268
00269
          io_uring_prep_readv(sqe, client_socket, &req->iov[0], req->iovec_count, 0);
00270
          io_uring_sqe_set_data(sqe, req);
00271
             (io_uring_submit(ring) < 0) {
           free(sqe);
00272
00273
            list_delete_node(&s->list, req);
00274
            res = ERROR_CODE;
00275
            continue;
00276
00277
          res = SUCCESS_CODE;
00278
00279
        pthread_mutex_unlock(&s->m_rings[sel]);
00280 }
00281
00282 static void add_read_continue(struct saurion *const s, struct request *oreq, const int sel) {
        pthread_mutex_lock(&s->m_rings[sel]);
00283
00284
        int res = ERROR_CODE;
00285
        while (res != SUCCESS_CODE) {
00286
          struct io_uring *ring = &s->rings[sel];
00287
          struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
00288
          while (!sge) {
00289
            sqe = io_uring_get_sqe(ring);
            usleep(TIMEOUT_RETRY);
00290
00291
          if (!set_request(&oreq, &s->list, oreq->prev_remain, NULL, 0)) {
00292
00293
            free(sqe);
            res = ERROR_CODE;
00294
00295
            continue;
00296
00297
          io_uring_prep_readv(sqe, oreq->client_socket, &oreq->iov[0], oreq->iovec_count, 0);
          io_uring_sqe_set_data(sqe, oreq);
if (io_uring_submit(ring) < 0) {</pre>
00298
00299
00300
            free (sqe);
00301
            list_delete_node(&s->list, oreq);
00302
            res = ERROR_CODE;
00303
            continue;
00304
00305
          res = SUCCESS CODE;
00306
00307
       pthread_mutex_unlock(&s->m_rings[sel]);
00308 }
00309
00310 static void add_write(struct saurion *const s, int fd, const char *const str, const int sel) {
00311
        int res = ERROR_CODE;
00312
        pthread_mutex_lock(&s->m_rings[sel]);
while (res != SUCCESS_CODE) {
00314
          struct io_uring *ring = &s->rings[sel];
00315
          struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
00316
          while (!sqe) {
00317
            sqe = io_uring_get_sqe(ring);
           usleep(TIMEOUT_RETRY);
00318
00319
00320
          struct request *req = NULL;
00321
          if (!set_request(&req, &s->list, strlen(str), (void *)str, 1)) {
00322
           free(sqe);
00323
            res = ERROR CODE;
00324
            continue;
00325
00326
          req->event_type = EVENT_TYPE_WRITE;
00327
          req->client_socket = fd;
00328
          io_uring_prep_writev(sqe, req->client_socket, req->iov, req->iovec_count, 0);
          io_uring_sqe_set_data(sqe, req);
if (io_uring_submit(ring) < 0) {</pre>
00329
00330
00331
            free(sqe);
00332
            list_delete_node(&s->list, req);
00333
            res = ERROR_CODE;
00334
            usleep(TIMEOUT_RETRY);
            continue;
00335
00336
00337
          res = SUCCESS CODE;
00338
00339
        pthread_mutex_unlock(&s->m_rings[sel]);
00340 }
00341
00343 static void handle_accept(const struct saurion *const s, const int fd) {
```

```
if (s->cb.on_connected) {
00345
          s->cb.on_connected(fd, s->cb.on_connected_arg);
00346
00347 }
00348
00349 [[nodiscard]]
00350 int read_chunk(void **dest, size_t *len, struct request *const req) {
00351
        // Initial checks
00352
        return ERROR_CODE;
}
         if (req->iovec_count == 0) {
00353
00354
00355
00356
        // Initizalization
00357
        size_t max_iov_cont = 0; //< Total size of request</pre>
00358
        for (size_t i = 0; i < req->iovec_count; ++i) {
00359
          max_iov_cont += req->iov[i].iov_len;
00360
00361
        size t cont sz = 0;
                                      //< Message content size
00362
        size_t cont_rem = 0;
                                     //< Remaining bytes of message content
00363
        size_t curr_iov = 0;
                                     //< IOVEC num currently reading
00364
        size_t curr_iov_off = 0; //< Offset in bytes of the current IOVEC</pre>
        size_t dest_off = 0;
void *dest_ptr = NULL; //< Write offset on the destiny array
void *dest_ptr = NULL; //< Destiny pointer, could be dest or prev</pre>
00365
00366
00367
        if (req->prev && req->prev_size && req->prev_remain) {
00368
          // There's a previous unfinished message
           cont_sz = req->prev_size;
00369
           cont_rem = req->prev_remain;
00370
           curr_iov = 0;
curr_iov_off = 0;
00371
00372
00373
           dest_off = cont_sz - cont_rem;
00374
           if (cont_rem <= max_iov_cont) {</pre>
00375
             *dest = req->prev;
             dest_ptr = *dest;
req->prev = NULL;
00376
00377
00378
             req->prev_size = 0;
00379
             req->prev_remain = 0;
00380
          } else {
00381
            dest_ptr = req->prev;
00382
             *dest = NULL;
00383
00384
        } else if (req->next_iov || req->next_offset) {
           // Reading the next message
00385
          curr_iov = req->next_iov;
curr_iov_off = req->next_offset;
00386
00387
           cont_sz = *((size_t *)(req->iov[curr_iov].iov_base + curr_iov_off));
00388
00389
           cont_sz = ntohll(cont_sz);
00390
           curr_iov_off += sizeof(uint64_t);
00391
           cont_rem = cont_sz;
          dest_off = cont_sz - cont_rem;
if ((curr_iov_off + cont_rem + 1) <= max_iov_cont) {</pre>
00392
00393
             *dest = malloc(cont_sz);
00394
00395
             dest_ptr = *dest;
00396
           } else {
00397
             reg->prev = malloc(cont sz);
             dest_ptr = req->prev;
00398
             *dest = NULL;
*len = 0;
00399
00400
00401
00402
        } else {
00403
          // Reading the first message
00404
           curr_iov = 0;
           curr_iov_off = 0;
00405
           cont_sz = *((size_t *)(req->iov[curr_iov].iov_base + curr_iov_off));
cont_sz = ntohll(cont_sz);
00406
00407
00408
           curr_iov_off += sizeof(uint64_t);
           cont_rem = cont_sz;
dest_off = cont_sz - cont_rem;
00409
00410
00411
           if (cont rem <= max iov cont) {
            *dest = malloc(cont_sz);
00412
00413
             dest_ptr = *dest;
00414
           } else {
            req->prev = malloc(cont_sz);
dest_ptr = req->prev;
00415
00416
             *dest = NULL;
00417
00418
          }
00419
00421
        size_t curr_iov_msg_rem = 0;
00422
00423
         // Copy loop
00424
        uint8 t ok = 1UL;
00425
        while (1) {
           curr_iov_msg_rem = MIN(cont_rem, (req->iov[curr_iov].iov_len - curr_iov_off));
00427
           memcpy(dest_ptr + dest_off, req->iov[curr_iov].iov_base + curr_iov_off, curr_iov_msg_rem);
00428
           dest_off += curr_iov_msg_rem;
00429
           curr_iov_off += curr_iov_msg_rem;
          cont_rem -= curr_iov_msg_rem;
if (cont_rem <= 0) {</pre>
00430
00431
```

```
// Finish reading
00433
            if (*((uint8_t *)(req->iov[curr_iov].iov_base + curr_iov_off)) != 0) {
00434
              ok = OUL;
00435
            *len = cont sz;
00436
00437
            ++curr_iov_off;
00438
            break;
00439
00440
          if (curr_iov_off >= (req->iov[curr_iov].iov_len)) {
00441
            ++curr iov;
00442
            if (curr_iov == req->iovec_count) {
00443
              break:
00444
00445
            curr_iov_off = 0;
00446
00447
00448
00449
        // Update status
00450
        if (req->prev) {
00451
          req->prev_size = cont_sz;
00452
          req->prev_remain = cont_rem;
00453
          *dest = NULL;
00454
          len = 0;
        } else {
00455
00456
          req->prev_size = 0;
00457
          req->prev_remain = 0;
00458
00459
        if (curr_iov < req->iovec_count) {
00460
          uint64_t next_sz = *(uint64_t *)(req->iov[curr_iov].iov_base + curr_iov_off);
          if ((req->iov[curr_iov].iov_len > curr_iov_off) && next_sz) {
    req->next_iov = curr_iov;
00461
00462
00463
            req->next_offset = curr_iov_off;
00464
00465
            req->next_iov = 0;
00466
            req->next_offset = 0;
00467
00468
        }
00469
00470
        // Finish
00471
        if (!ok) {
         // Esto solo es posible si no se encuentra un 0 al final del la lectura // buscar el siguiente 0 y \dots probar fortuna
00472
00473
00474
          free(dest_ptr);
00475
          dest_ptr = NULL;
          *dest = NULL;
*len = 0;
00476
00477
00478
          req->next_iov = 0;
00479
          req->next_offset = 0;
          for (size_t i = curr_iov; i < req->iovec_count; ++i) {
00480
            for (size_t j = curr_iov_off; j < req->iov[i].iov_len; ++j) {
00481
              uint8_t foot = *(uint8_t *)(req->iov[i].iov_base + j);
00482
00483
              if (foot == 0) {
00484
                req->next_iov = i;
                req->next_offset = (j + 1) % req->iov[i].iov_len;
00485
00486
                return ERROR_CODE;
00487
              }
            }
00488
00489
00490
          return ERROR_CODE;
00491
00492
        return SUCCESS CODE:
00493 }
00494
00495 static void handle_read(struct saurion *const s, struct request *const req) {
00496
        void *msg = NULL;
00497
        size_t len = 0;
00498
        while (1) {
         if (!read_chunk(&msg, &len, req)) {
00499
00500
            break;
00501
00502
          // Hay siguiente
00503
          if (req->next_iov || req->next_offset) {
00504
            if (s->cb.on_readed && msg) {
              s->cb.on_readed(req->client_socket, msg, len, s->cb.on_readed_arg);
00505
00506
00507
            free (msg);
00508
            msg = NULL;
00509
            continue;
00510
00511
          // Hay previo pero no se ha completado
          if (req->prev && req->prev_size && req->prev_remain) {
00512
            add_read_continue(s, req, next(s));
00514
00515
00516
          // Hay un único mensaje y se ha completado
          if (s->cb.on readed && msg) {
00517
00518
            s->cb.on_readed(req->client_socket, msg, len, s->cb.on_readed_arg);
```

```
00519
00520
          free (msg);
00521
         msg = NULL;
00522
         break;
00523
00524
       add read(s, reg->client socket);
00525 }
00526
00527 static void handle_write(const struct saurion *const s, const int fd) {
00528
       if (s->cb.on wrote) {
00529
         s->cb.on_wrote(fd, s->cb.on_wrote_arg);
00530
00531 }
00532
00533 static void handle_error(const struct saurion *const s, const struct request *const req) {
       if (s->cb.on_error) {
  const char *resp = "ERROR";
00534
00535
00536
         s->cb.on_error(req->client_socket, resp, (ssize_t)strlen(resp), s->cb.on_error_arg);
00537
00538 }
00539
00540 static void handle_close(const struct saurion *const s, const struct request *const req) {
00541 if (s->cb.on_closed) {
         s->cb.on_closed(req->client_socket, s->cb.on_closed_arg);
00542
00543
00544
       close(req->client_socket);
00545 }
00546
00548 int EXTERNAL_set_socket(const int p) {
       int sock = 0;
00549
00550
       struct sockaddr_in srv_addr;
00551
00552
       sock = socket(PF_INET, SOCK_STREAM, 0);
00553
       if (sock < 1) {</pre>
00554
         return ERROR_CODE;
00555
00556
00557
       int enable = 1;
00558
       if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(int)) < 0) {</pre>
00559
         return ERROR_CODE;
00560
00561
       if (setsockopt(sock, SOL_SOCKET, SO_REUSEPORT, &enable, sizeof(int)) < 0) {</pre>
00562
         return ERROR_CODE;
00563
00564
00565
       memset(&srv_addr, 0, sizeof(srv_addr));
       srv_addr.sin_family = AF_INET;
srv_addr.sin_port = htons(p);
00566
00567
00568
       srv addr.sin addr.s addr = htonl(INADDR ANY);
00569
00570
       /\star We bind to a port and turn this socket into a listening
00571 * socket.
00572 * */
00573
       if (bind(sock, (const struct sockaddr *)&srv_addr, sizeof(srv_addr)) < 0) {</pre>
00574
         return ERROR CODE;
00575
00576
00577
       return ERROR_CODE;
}
       if (listen(sock, ACCEPT_QUEUE) < 0) {</pre>
00578
00579
00580
00581
       return sock;
00582 }
00583
00584 [[nodiscard]]
00585 struct saurion *saurion\_create(uint32\_t n\_threads) {
00586
       // Asignar memoria
00587
       struct saurion *p = (struct saurion *)malloc(sizeof(struct saurion));
00588
       if (!p) {
00589
         return NULL;
00590
00591
       // Inicializar mutex
00592
       int ret = 0;
00593
       ret = pthread_mutex_init(&p->status_m, NULL);
       if (ret) {
00594
00595
         free(p);
00596
         return NULL;
00597
00598
       ret = pthread cond init(&p->status c, NULL);
       if (ret) {
00599
00600
         free(p);
00601
00602
00603
       p->m_rings = (pthread_mutex_t *)malloc(n_threads * sizeof(pthread_mutex_t));
00604
       if (!p->m_rings) {
00605
         free(p);
```

```
return NULL;
00607
00608
        for (uint32_t i = 0; i < n_threads; ++i) {</pre>
00609
          pthread_mutex_init(&(p->m_rings[i]), NULL);
00610
00611
        // Inicializar miembros
        p->ss = 0;
00612
00613
        n_threads = (n_threads < 2 ? 2 : n_threads);</pre>
00614
        n_threads = (n_threads > NUM_CORES ? NUM_CORES : n_threads);
        p->n_threads = n_threads;
p->status = 0;
00615
00616
        p->list = NULL;
00617
00618
        p->cb.on_connected = NULL;
00619
        p->cb.on_connected_arg = NULL;
00620
        p->cb.on_readed = NULL;
00621
        p->cb.on_readed_arg = NULL;
        p->cb.on_wrote = NULL;
00622
        p->cb.on_wrote_arg = NULL;
00623
        p->cb.on_closed = NULL;
00624
00625
        p->cb.on_closed_arg = NULL;
00626
        p->cb.on_error = NULL;
00627
        p->cb.on_error_arg = NULL;
        p->next = 0;
00628
        // Inicializar efds
00629
00630
        p->efds = (int *)malloc(sizeof(int) * p->n_threads);
        if (!p->efds) {
00631
          free(p->m_rings);
00632
00633
          free(p);
00634
          return NULL;
00635
        for (uint32_t i = 0; i < p->n_threads; ++i) {
  p->efds[i] = eventfd(0, EFD_NONBLOCK);
  if (p->efds[i] == ERROR_CODE) {
00636
00637
00638
00639
             for (uint32_t j = 0; j < i; ++j) {
00640
              close(p->efds[j]);
00641
00642
             free (p->efds);
             free(p->m_rings);
00643
00644
             free(p);
00645
            return NULL;
00646
00647
        // Inicializar rings
00648
00649
        p->rings = (struct io_uring *)malloc(sizeof(struct io_uring) * p->n_threads);
00650
        if (!p->rings) {
00651
          for (uint32_t j = 0; j < p->n_threads; ++j) {
00652
            close(p->efds[j]);
00653
00654
          free (p->efds);
          free(p->m_rings);
00655
00656
          free(p);
00657
          return NULL;
00658
        for (uint32_t i = 0; i < p->n_threads; ++i) {
  memset(&p->rings[i], 0, sizeof(struct io_uring));
00659
00660
00661
          ret = io_uring_queue_init(SAURION_RING_SIZE, &p->rings[i], 0);
          if (ret) {
00663
            for (uint32_t j = 0; j < p->n_threads; ++j) {
00664
              close(p->efds[j]);
00665
00666
             free (p->efds):
             free(p->rings);
00667
00668
             free(p->m_rings);
00669
             free(p);
00670
             return NULL;
00671
          }
00672
00673
        p->pool = ThreadPool_create(p->n_threads);
00674
        return p:
00675 }
00676
00677 void saurion_worker_master(void *arg) {
00678 struct saurion *const s = (struct saurion *)arg;
00679
        struct io_uring ring = s->rings[0];
        struct io_uring_cqe *cqe = NULL;
00680
00681
        struct sockaddr_in client_addr;
00682
        socklen_t client_addr_len = sizeof(client_addr);
00683
00684
        add_efd(s, s->efds[0], 0);
00685
        add_accept(s, &client_addr, &client_addr_len);
00686
00687
        pthread_mutex_lock(&s->status_m);
00688
        s->status = 1;
00689
        pthread_cond_signal(&s->status_c);
00690
        pthread_mutex_unlock(&s->status_m);
00691
        while (1) {
00692
          int ret = io uring wait cge(&ring, &cge);
```

```
if (ret < 0) {</pre>
00694
           free(cqe);
00695
            return;
00696
          struct request *req = (struct request *)cqe->user_data;
00697
00698
          if (!rea) {
00699
            io_uring_cqe_seen(&s->rings[0], cqe);
00700
00701
00702
          if (cqe->res < 0) {
00703
            list_delete_node(&s->list, req);
00704
            return:
00705
00706
          if (req->client_socket == s->efds[0]) {
00707
            io_uring_cqe_seen(&s->rings[0], cqe);
00708
            list_delete_node(&s->list, req);
00709
            break:
00710
00711
          /* Mark this request as processed */
00712
          io_uring_cqe_seen(&s->rings[0], cqe);
00713
          switch (req->event_type) {
00714
            case EVENT_TYPE_ACCEPT:
00715
              handle_accept(s, cqe->res);
00716
              add_accept(s, &client_addr, &client_addr_len);
              add_read(s, cqe->res);
list_delete_node(&s->list, req);
00717
00718
00719
00720
            case EVENT_TYPE_READ:
00721
              if (cqe->res < 0) {</pre>
               handle_error(s, req);
00722
00723
00724
              if (cqe->res < 1) {
00725
               handle_close(s, req);
00726
00727
              if (cqe->res > 0) {
00728
                handle_read(s, req);
00729
00730
              list_delete_node(&s->list, req);
00731
              break;
00732
            case EVENT_TYPE_WRITE:
00733
              handle_write(s, req->client_socket);
00734
              list_delete_node(&s->list, req);
00735
              break:
00736
         }
00737
00738
       pthread_mutex_lock(&s->status_m);
00739
        s->status = 2;
00740
        pthread_cond_signal(&s->status_c);
00741
        pthread_mutex_unlock(&s->status_m);
00742
        return:
00743 }
00744
00745 void saurion_worker_slave(void *arg) {
00746
       struct saurion_wrapper *const ss = (struct saurion_wrapper *)arg;
00747
        struct saurion *s = ss->s;
00748
        const int sel = ss->sel;
00749
       free(ss);
00750
        struct io_uring ring = s->rings[sel];
00751
       struct io_uring_cqe *cqe = NULL;
00752
00753
       add efd(s, s->efds[sel], sel);
00754
00755
       pthread_mutex_lock(&s->status_m);
00756
        s->status = 1;
00757
        pthread_cond_signal(&s->status_c);
00758
        pthread_mutex_unlock(&s->status_m);
00759
        while (1) {
00760
          int ret = io_uring_wait_cqe(&ring, &cqe);
          if (ret < 0) {</pre>
00761
00762
            free(cqe);
00763
00764
00765
          struct request *req = (struct request *)cqe->user_data;
00766
          if (!rea) {
00767
            io_uring_cqe_seen(&ring, cqe);
00768
            continue;
00769
00770
          if (cqe->res < 0) {</pre>
00771
            list_delete_node(&s->list, req);
00772
            return:
00773
00774
          if (req->client_socket == s->efds[sel]) {
00775
            io_uring_cqe_seen(&ring, cqe);
00776
            list_delete_node(&s->list, req);
00777
            break;
00778
00779
          /* Mark this request as processed */
```

```
io_uring_cqe_seen(&ring, cqe);
00781
          switch (req->event_type) {
00782
            case EVENT_TYPE_READ:
00783
             if (cqe->res < 0) {
00784
                handle_error(s, req);
00785
00786
              if (cqe->res < 1) {
00787
                handle_close(s, req);
00788
              if (cqe->res > 0) {
00789
00790
               handle_read(s, req);
00791
00792
              list delete node(&s->list, reg);
00793
              break;
00794
            case EVENT_TYPE_WRITE:
00795
             handle_write(s, req->client_socket);
00796
              list_delete_node(&s->list, req);
00797
              break;
00798
         }
00799
00800
        pthread_mutex_lock(&s->status_m);
00801
        s->status = 2;
        pthread_cond_signal(&s->status_c);
00802
00803
        pthread_mutex_unlock(&s->status_m);
00804
        return;
00805 }
00806
00807 [[nodiscard]]
00808 int saurion_start(struct saurion *const s) {
00809
        pthread_mutex_init(&print_mutex, NULL);
00810
        ThreadPool_init(s->pool);
00811
        ThreadPool_add_default(s->pool, saurion_worker_master, s);
        struct saurion_wrapper *ss = NULL;
for (uint32_t i = 1; i < s->n_threads; ++i) {
00812
00813
00814
         ss = (struct saurion_wrapper *)malloc(sizeof(struct saurion_wrapper));
          ss->s = s;
00815
00816
          ss->sel = i;
          ThreadPool_add_default(s->pool, saurion_worker_slave, ss);
00818
00819
        return SUCCESS_CODE;
00820 }
00821
00822 void saurion_stop(const struct saurion *const s) { 00823    uint64_t u = 1;
        for (uint32_t i = 0; i < s->n_threads; ++i)
00825
         while (write(s->efds[i], &u, sizeof(u)) < 0) {</pre>
00826
           usleep(TIMEOUT_RETRY);
00827
00828
00829
        ThreadPool_wait_empty(s->pool);
00830 }
00831
00832 void saurion_destroy(struct saurion *const s) {
00833
        pthread_mutex_lock(&s->status_m);
00834
        while (s->status == 1) {
00835
         pthread_cond_wait(&s->status_c, &s->status_m);
00836
00837
        pthread_mutex_unlock(&s->status_m);
00838
        ThreadPool_destroy(s->pool);
00839
        for (uint32_t i = 0; i < s->n_threads; ++i) {
         io_uring_queue_exit(&s->rings[i]);
00840
00841
         pthread_mutex_destroy(&s->m_rings[i]);
00842
00843
00844
        list_free(&s->list);
00845
        for (uint32_t i = 0; i < s->n_threads; ++i) {
00846
         close(s->efds[i]);
00847
00848
        free(s->efds);
        if (!s->ss) {
00849
00850
         close(s->ss);
00851
00852
       free(s->rings);
00853
        pthread_mutex_destroy(&s->status_m);
00854
        pthread_cond_destroy(&s->status_c);
00855
        free(s);
00856 }
00857
00858 void saurion_send(struct saurion *const s, const int fd, const char *const msg) {
00859
       add_write(s, fd, msg, next(s));
00860 }
```

### 7.19 / w/saurion/saurion/src/saurion.cpp File Reference

```
#include "saurion.hpp"
#include <cstdint>
#include "low_saurion.h"
Include dependency graph for saurion.cpp:
```

### 7.20 saurion.cpp

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

```
00001 #include "saurion.hpp'
00002
00003 #include <cstdint>
00004
00005 #include "low_saurion.h"
00006
00007 Saurion::Saurion(const uint32 t thds, const int sck) noexcept {
00008 this->s = saurion_create(thds);
00009
        if (!this->s) {
00010
         return;
00011
00012
       this->s->ss = sck;
00013 }
00014
00015 Saurion::~Saurion() { saurion_destroy(this->s); }
00016
00017 void Saurion::init() noexcept {
00018
       if (!saurion_start(this->s)) {
00019
         return;
00020
00021 }
00023 void Saurion::stop() noexcept { saurion_stop(this->s); }
00024
00025 Saurion *Saurion::on_connected(Saurion::ConnectedCb ncb, void *arg) noexcept {
00026 s->cb.on_connected = ncb;
00027 s->cb.on_connected_arg =
       s->cb.on_connected_arg = arg;
00028
       return this;
00029 }
00030
00031 Saurion *Saurion::on_readed(Saurion::ReadedCb ncb, void *arg) noexcept {
00032 s->cb.on_readed = ncb;
00033
       s->cb.on_readed_arg = arg;
00034
        return this;
00035 }
00036
00037 Saurion *Saurion::on_wrote(Saurion::WroteCb ncb, void *arg) noexcept {
00038 s->cb.on_wrote = ncb;
00039 s->cb.on_wrote_arg =
       s->cb.on_wrote_arg = arg;
00040
        return this;
00041 }
00042
00043 Saurion *Saurion::on_closed(Saurion::ClosedCb ncb, void *arg) noexcept {
00044 s->cb.on_closed = ncb;
        s->cb.on_closed_arg = arg;
00045
00046 return this;
00047 }
00048
00049 Saurion *Saurion::on_error(Saurion::ErrorCb ncb, void *arg) noexcept {
00050 s \rightarrow cb.on\_error = ncb;
00051
        s->cb.on_error_arg = arg;
00052
       return this;
00053 }
00054
00055 void Saurion::send(const int fd, const char *const msg) noexcept { saurion_send(this->s, fd, msg); }
```

# 7.21 /\_\_w/saurion/saurion/src/threadpool.cpp File Reference

```
#include "threadpool.hpp"
#include <pthread.h>
#include <stdexcept>
Include dependency graph for threadpool.cpp:
```

7.22 threadpool.cpp 107

### **Typedefs**

```
    using TP = ThreadPool
```

using T = Task

### 7.21.1 Typedef Documentation

#### 7.21.1.1 T

```
using T = Task
```

Definition at line 12 of file threadpool.cpp.

#### 7.21.1.2 TP

```
using TP = ThreadPool
```

Definition at line 9 of file threadpool.cpp.

## 7.22 threadpool.cpp

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

```
00001 // This is a personal academic project. Dear PVS-Studio, please check it.
00002 // PVS-Studio Static Code Analyzer for C, C++, C#, and Java: https://pvs-studio.com 00003 #include "threadpool.hpp"
00004
00005 #include <pthread.h>
00006
00007 #include <stdexcept>
80000
00009 using TP = ThreadPool;
00010
00011 //****** Task ********
00012 using T = Task;
00013 T:: Task (void (*nfn) (void*), void* narg) : function(nfn), argument(narg) {}
00014
00015 //****** AsyncQueue ********
00016 TP::AsyncQueue::AsyncQueue(uint32_t cnt) : m_max(cnt), m_cnt(0) {}
00017
00018 TP::AsyncQueue::~AsyncQueue() { pthread_mutex_destroy(&m_mtx); }
00019
00020 void TP::AsyncQueue::push(Task* task) {
00021 pthread_mutex_lock(&m_mtx);
00022
        m_queue.push(task);
00023 pthread_mutex_unlock(&m_mtx);
00024 }
00025 Task* TP::AsyncQueue::front()
00026 pthread_mutex_lock(&m_mtx);
00027
        if ((m_cnt >= m_max) && (m_max != 0)) {
       pthread_mutex_unlock(&m_mtx);
throw std::out_of_range("reached max parallel tasks");
00028
00029
00030
00031
       Task* task = m_queue.front();
00032
       m_queue.pop();
00033
        ++m_cnt;
00034
        pthread_mutex_unlock(&m_mtx);
00035
        return task;
00036 }
00037 void TP::AsyncQueue::pop() {
```

```
pthread_mutex_lock(&m_mtx);
00039
00040
       pthread_mutex_unlock(&m_mtx);
00041 }
00042
00043 bool TP::AsyncQueue::empty() {
00044 pthread_mutex_lock(&m_mtx);
00045
        bool empty = m_queue.empty();
00046
       pthread_mutex_unlock(&m_mtx);
00047
        return empty;
00048 }
00049
00050 //***** AsyncMultiQueue *******
00051
00052 TP::AsyncMultiQueue::AsyncMultiQueue() {
00053 new_queue(0, 0);
00054
       m_it = m_queues.begin();
00055 }
00056 TP::AsyncMultiQueue::~AsyncMultiQueue() {
00057
       for (auto& queue : m_queues) {
00058
         delete queue.second;
00059
00060 }
00061
00062 void TP::AsyncMultiQueue::new_queue(uint32_t gid, uint32_t cnt) {
      if (m_queues.find(qid) != m_queues.end()) {
00064
          throw std::out_of_range("queue already exists");
00065
00066
       m_queues.emplace(qid, new TP::AsyncQueue(cnt));
00067 }
00068 void TP::AsyncMultiQueue::remove_queue(uint32_t qid) {
       auto queue = m_queues.find(qid);
00070
       if (queue == m_queues.end()) {
00071
         throw std::out_of_range("queue not found");
00072
00073
       delete queue->second;
00074
       m_queues.erase(qid);
00075 }
00076
00077 void TP::AsyncMultiQueue::push(uint32_t qid, void (*nfn)(void*), void* arg) {
00078
       m_queues.at(qid)->push(new Task{nfn, arg});
00079 }
00080 Task* TP::AsyncMultiQueue::front(uint32 t& gid) {
00081
       if (empty()) {
         throw std::out_of_range("empty queue");
00082
00083
00084
       auto newit = m_it;
00085
        ++newit;
        while (newit != m_it) {
00086
        if (newit == m_queues.end()) {
00087
00088
           newit = m_queues.begin();
00089
00090
         if (!newit->second->empty()) {
00091
           break;
00092
00093
         ++newit;
00094
00095
        Task* task = newit->second->front();
        qid = newit->first;
00096
00097
        m_{it} = newit;
       return task:
00098
00099 }
00100 void TP::AsyncMultiQueue::pop(uint32_t qid) { m_queues.at(qid)->pop(); }
00101 void TP::AsyncMultiQueue::clear() {
00102
        if (empty()) {
00103
         return;
00104
       auto newit = m_queues.begin();
00105
       while (newit != m_queues.end()) {
00106
        while (!newit->second->empty())
00107
00108
          delete newit->second->front();
00109
           newit->second->pop();
00110
00111
          ++newit;
00112
       }
00113 }
00114
00115 bool TP::AsyncMultiQueue::empty() {
       for (auto& queue : m_queues) {
  if (!queue.second->empty()) {
00116
00117
00118
           return false;
00119
         }
00120
00121
        return true;
00122 }
00123
00124 //****** ThreadPool ********
```

7.22 threadpool.cpp 109

```
00126 pthread_mutex_t TP::s_mtx = PTHREAD_MUTEX_INITIALIZER;
00127
00128 TP::ThreadPool() : ThreadPool(4) {}
00129
00130 TP::ThreadPool(size_t num_threads)
         : m_nth(num_threads < 2 ? 2 : num_threads),
00132
           m_started(0),
00133
           m_ths(new pthread_t[m_nth]{0}),
00134
           m_fstop(0)
00135
          m_faccept(0) {}
00136
00137 void TP::init() {
00138 if (m_started != 0) {
00139
         return;
00140
00141
       m_faccept = 1;
       m_fstop = 0;
00142
00143
       pthread_mutex_lock(&s_mtx);
00144
        for (size_t i = 0; i < m_nth; ++i) {</pre>
00145
         pthread_create(&m_ths[i], nullptr, thread_entry, this);
00146
          ++m_started;
00147
00148
       pthread_mutex_unlock(&s_mtx);
00149 }
00150 void TP::stop() {
00151
       if (m_started == 0) {
00152
         return;
00153
00154
       m_fstop = 0;
00155
       m faccept = 0:
00156
       wait_empty();
00157
       pthread_mutex_lock(&m_q_mtx);
00158
00159
        m_fstop = 1;
        pthread_cond_broadcast(&m_q_cond);
00160
00161
        pthread_mutex_unlock(&m_q_mtx);
00162
        // Detener los hilos
00163
        for (size_t i = 0; i < m_started; ++i) {</pre>
00164
         pthread_join(m_ths[i], nullptr);
00165
00166
       m_started = 0;
       m_nth = 0;
00167
00168 }
00169 void TP::add(uint32_t qid, void (*nfn)(void*), void* arg) {
00170
        if (m_faccept == 0) {
00171
         throw std::logic_error("threadpool already closed");
00172
00173
        if (nfn == nullptr) {
00174
         throw std::logic_error("function pointer cannot be null");
00175
00176
        bool failed = false;
00177
        pthread_mutex_lock(&m_q_mtx);
00178
        try {
00179
         m_queues.push(qid, nfn, arg);
        } catch (const std::out_of_range& e) {
00180
         failed = true;
00182
00183
        pthread_cond_signal(&m_q_cond);
00184
        pthread_mutex_unlock(&m_q_mtx);
00185
        if (failed) {
00186
         throw std::out_of_range("queue not found");
00187
00188 }
00189 void TP::add(void (*nfn)(void*), void* arg) {
00190
       add(0, nfn, arg); // Agregar a la cola por defecto
00191 }
00192 void TP::new_queue(uint32_t qid, uint32_t cnt) {
00193
       pthread mutex lock(&m g mtx);
00194
        try {
00195
         m_queues.new_queue(qid, cnt);
00196
        } catch (const std::out_of_range& e) {
        pthread_mutex_unlock(&m_q_mtx);
00197
00198
         throw e:
00199
00200
       pthread_mutex_unlock(&m_q_mtx);
00201 }
00202 void TP::remove_queue(uint32_t qid) {
        if (qid != 0) {
00203
         bool failed = false;
00204
00205
          pthread_mutex_lock(&m_q_mtx);
00206
          try {
00207
           m_queues.remove_queue(qid);
00208
          } catch (const std::out_of_range& e) {
00209
           failed = true;
00210
00211
          pthread cond signal (&m g cond);
```

```
pthread_mutex_unlock(&m_q_mtx);
00213
           if (failed) {
00214
             throw std::out_of_range("queue not found");
00215
00216
00217 }
00218 bool TP::empty() { return m_queues.empty(); }
00219 void TP::wait_closeable() {
00220 pthread_mutex_lock(&m_q_mtx);
        while (empty() && m_fstop == 0) {
  pthread_cond_wait(&m_q_cond, &m_q_mtx);
00221
00222
00223
00224
        pthread_mutex_unlock(&m_q_mtx);
00225 }
00226 void TP::wait_empty() {
00227
        pthread_mutex_lock(&m_q_mtx);
        pthread_cond_wait(&m_q_cond, &m_q_mtx);
}
00228
00229
00231
        pthread_mutex_unlock(&m_q_mtx);
00232 }
00233 TP::~ThreadPool() {
00234
        stop();
00235
        m_queues.clear();
00236
        delete[] m_ths;
00237
        pthread_mutex_destroy(&s_mtx);
00238 }
00239
00240 void TP::thread_worker() {
00241
        // Lógica del trabajador del hilo
uint32_t qid = 0;
00242
00243
        while (m_fstop == 0) {
00244
         // Buscar una tarea para ejecutar
00245
00246
             {\tt pthread\_mutex\_lock(\&m\_q\_mtx);}
             Task* task = m_queues.front(qid);
pthread_cond_signal(&m_q_cond);
00247
00248
             pthread_mutex_unlock(&m_q_mtx);
00250
             try {
00251
              task->function(task->argument);
             } catch (...) {
00252
00253
             delete task; // TODO delete task
00254
             pthread_mutex_lock(&m_q_mtx);
00255
00256
             m_queues.pop(qid);
00257
             pthread_cond_broadcast(&m_q_cond);
00258
             pthread_mutex_unlock(&m_q_mtx);
00259
           } catch (const std::out_of_range& e) {
             pthread_mutex_unlock(&m_q_mtx);
00260
00261
             wait closeable();
00262
00263 }
00264 }
00265 void* TP::thread_entry(void* arg) {
00266
        auto* pool = static_cast<TP*>(arg);
pool->thread_worker();
00267
        return nullptr;
00269 }
```

# Index

```
/_w/saurion/saurion/include/cthreadpool.hpp, 63, 66
                                                           request, 34
                                                       ClosedCb
/ w/saurion/saurion/include/linked list.h, 66, 67
  w/saurion/saurion/include/low saurion.h, 67, 73
                                                           Saurion, 39
/ w/saurion/saurion/include/low saurion secret.h, 74
                                                       ConnectedCb
/_w/saurion/saurion/include/saurion.hpp, 75
                                                           Saurion, 39
/ w/saurion/saurion/include/threadpool.hpp, 76
                                                       create node
/ w/saurion/saurion/src/cthreadpool.cpp, 78, 81
                                                           linked list.c, 81
/ w/saurion/saurion/src/linked list.c, 81, 84
                                                       cthreadpool.cpp
/_w/saurion/saurion/src/low_saurion.c, 86, 96
                                                           ThreadPool_add, 78
                                                           ThreadPool_add_default, 78
  w/saurion/saurion/src/saurion.cpp, 106
  w/saurion/saurion/src/threadpool.cpp, 106, 107
                                                           ThreadPool create, 79
\simAsyncMultiQueue
                                                           ThreadPool create default, 79
    ThreadPool::AsyncMultiQueue, 26
                                                           ThreadPool_destroy, 79
~AsyncQueue
                                                           ThreadPool_empty, 79
    ThreadPool::AsyncQueue, 30
                                                           ThreadPool init, 79
\simSaurion
                                                           ThreadPool new queue, 80
                                                           ThreadPool_remove_queue, 80
    Saurion, 40
\simTask
                                                           ThreadPool_stop, 80
                                                           ThreadPool_wait_empty, 80
     Task, 52
\simThreadPool
                                                       cthreadpool.hpp
    ThreadPool, 56
                                                           ThreadPool, 63
                                                           ThreadPool add, 64
add
                                                           ThreadPool add default, 64
    ThreadPool, 56
                                                           ThreadPool create, 64
add accept
                                                           ThreadPool create default, 64
    low saurion.c, 88
                                                           ThreadPool destroy, 64
add efd
                                                           ThreadPool empty, 65
    low_saurion.c, 89
                                                           ThreadPool init, 65
add read
                                                           ThreadPool_new_queue, 65
    low saurion.c, 90
                                                           ThreadPool remove queue, 65
add_read_continue
                                                           ThreadPool stop, 65
    low_saurion.c, 90
                                                           ThreadPool_wait_empty, 66
add write
    low saurion.c, 91
                                                       efds
allocate iovec
                                                           low saurion.h, 68
    LowSaurion, 12
                                                           saurion, 36
argument
                                                       empty
    Task, 53
                                                           ThreadPool, 57
AsyncMultiQueue
                                                           ThreadPool::AsyncMultiQueue, 26
    ThreadPool, 55
                                                           ThreadPool::AsyncQueue, 30
    ThreadPool::AsyncMultiQueue, 25, 26
                                                       ErrorCb
AsyncQueue
                                                           Saurion, 40
    ThreadPool, 55
                                                       event_type
    ThreadPool::AsyncQueue, 29, 30
                                                           request, 34
                                                       EVENT TYPE ACCEPT
children
                                                           low_saurion.c, 87
    Node, 33
                                                       EVENT_TYPE_ERROR
clear
                                                           low saurion.c, 87
    ThreadPool::AsyncMultiQueue, 26
                                                       EVENT TYPE READ
client_socket
                                                           low_saurion.c, 87
```

EVENT_TYPE_WAIT	list_insert
low_saurion.c, 88	linked_list.c, 83
EVENT_TYPE_WRITE	list_mutex
low_saurion.c, 88	linked list.c, 84
EXTERNAL_set_socket	low saurion.c
LowSaurion, 12	add_accept, 88
	add_efd, 89
FORCE	add_read, 90
threadpool.hpp, 76	add_read_continue, 90
free node	add_read_continue, 90
linked_list.c, 82	<del>-</del> '
free_request	EVENT_TYPE_ACCEPT, 87
LowSaurion, 13	EVENT_TYPE_ERROR, 87
	EVENT_TYPE_READ, 87
front	EVENT_TYPE_WAIT, 88
ThreadPool::AsyncMultiQueue, 27	EVENT_TYPE_WRITE, 88
ThreadPool::AsyncQueue, 30	handle_accept, 91
function	handle_close, 91
Task, 53	handle_error, 92
	handle read, 92
handle_accept	handle write, 92
low_saurion.c, 91	htonll, 93
handle_close	MAX, 88
low_saurion.c, 91	MIN, 88
handle_error	next, 93
low_saurion.c, 92	
handle_read	ntohll, 93
low_saurion.c, 92	print_mutex, 95
handle_write	saurion_worker_master, 93
low_saurion.c, 92	saurion_worker_slave, 94
htonll	low_saurion.h
	efds, 68
low_saurion.c, 93	list, 68
init	m_rings, 68
-	n_threads, 69
Saurion, 41	next, 69
ThreadPool, 57	on_closed, 69
initialize_iovec	on_closed_arg, 69
LowSaurion, 13	on connected, 70
iov	on_connected_arg, 70
request, 34	on_error, 70
iovec_count	on_error_arg, 71
request, 34	on_readed, 71
KINDLY	on_readed_arg, 71
threadpool.hpp, 76	on_wrote, 71
	on_wrote_arg, 72
linked_list.c	pool, 72
create_node, 81	rings, 72
free_node, 82	ss, 72
list_delete_node, 82	status, 72
list free, 83	status_c, 73
list_insert, 83	status_m, 73
list mutex, 84	LowSaurion, 9
list	allocate_iovec, 12
	EXTERNAL_set_socket, 12
low_saurion.h, 68	free_request, 13
saurion, 36	initialize_iovec, 13
list_delete_node	
linked_list.c, 82	PACKING_SZ, 12
list_free	read_chunk, 15
linked_list.c, 83	saurion_create, 18

saurion_destroy, 19	ptr, 33
saurion_send, 20	size, 33
saurion_start, 20	ntohll
saurion_stop, 21	low_saurion.c, 93
set_request, 21	
	on_closed
m_cnt	low_saurion.h, 69
ThreadPool::AsyncQueue, 32	Saurion, 41
m_faccept	saurion::saurion_callbacks, 44
ThreadPool, 60	saurion_callbacks, 48
m_fstop	on_closed_arg
ThreadPool, 60	low_saurion.h, 69
m_it	saurion::saurion_callbacks, 45
ThreadPool::AsyncMultiQueue, 28	saurion_callbacks, 48
m_max	on_connected
ThreadPool::AsyncQueue, 32	low_saurion.h, 70
m mtx	Saurion, 41
ThreadPool::AsyncQueue, 32	saurion::saurion_callbacks, 45
m nth	saurion_callbacks, 48
ThreadPool, 60	on_connected_arg
m_q_cond	low_saurion.h, 70
ThreadPool, 60	saurion::saurion_callbacks, 45
m_q_mtx	saurion_callbacks, 49
ThreadPool, 60	on error
	low_saurion.h, 70
m_queue ThreadPeal::AgynaQueue, 33	
ThreadPool::AsyncQueue, 32	Saurion, 42
m_queues ThreadPool 61	saurion::saurion_callbacks, 45
ThreadPool, 61	saurion_callbacks, 49
ThreadPool::AsyncMultiQueue, 29	on_error_arg
m_rings	low_saurion.h, 71
low_saurion.h, 68	saurion::saurion_callbacks, 46
saurion, 37	saurion_callbacks, 49
m_started	on_readed
ThreadPool, 61	low_saurion.h, 71
m_ths	Saurion, 42
ThreadPool, 61	saurion::saurion_callbacks, 46
MAX	saurion_callbacks, 49
low_saurion.c, 88	on_readed_arg
MIN	low_saurion.h, 71
low_saurion.c, 88	saurion::saurion_callbacks, 46
	saurion_callbacks, 50
n_threads	on_wrote
low_saurion.h, 69	low_saurion.h, 71
saurion, 37	Saurion, 42
new_queue	saurion::saurion_callbacks, 46
ThreadPool, 57	saurion_callbacks, 50
ThreadPool::AsyncMultiQueue, 27	on_wrote_arg
next	low_saurion.h, 72
low_saurion.c, 93	saurion::saurion_callbacks, 47
low_saurion.h, 69	saurion_callbacks, 50
Node, 33	operator=
saurion, 37	Saurion, 42, 43
next_iov	Task, 52, 53
request, 35	ThreadPool, 57, 58
next offset	ThreadPool::AsyncMultiQueue, 27
request, 35	ThreadPool::AsyncQueue, 31
Node, 32	imeaur oom.haynoqueue, 31
children, 33	PACKING_SZ
next, 33	LowSaurion, 12
,	

pool	s, 43
low_saurion.h, 72	Saurion, 40, 41
saurion, 37	send, 43
pop	stop, 43
ThreadPool::AsyncMultiQueue, 28	WroteCb, 40
ThreadPool::AsyncQueue, 31	saurion, 36
prev	efds, 36
request, 35	list, 36
prev_remain	m_rings, 37
request, 35	n threads, 37
prev_size	next, 37
request, 35	pool, 37
print mutex	rings, 37
low_saurion.c, 95	ss, 38
ptr	status, 38
Node, 33	status_c, 38
push	status_m, 38
•	saurion::saurion callbacks, 44
ThreadPool::AsyncMultiQueue, 28	<del>-</del>
ThreadPool::AsyncQueue, 31	on_closed, 44
road ohunk	on_closed_arg, 45
read_chunk	on_connected, 45
LowSaurion, 15	on_connected_arg, 45
ReadedCb	on_error, 45
Saurion, 40	on_error_arg, 46
remove_queue	on_readed, 46
ThreadPool, 58	on_readed_arg, 46
ThreadPool::AsyncMultiQueue, 28	on_wrote, 46
request, 34	on_wrote_arg, 47
client_socket, 34	saurion_callbacks, 47
event_type, 34	on_closed, 48
iov, 34	on_closed_arg, 48
iovec_count, 34	on_connected, 48
next_iov, 35	on_connected_arg, 49
next_offset, 35	on_error, 49
prev, 35	on_error_arg, 49
prev remain, 35	on_readed, 49
prev_size, 35	on_readed_arg, 50
rings	on_wrote, 50
low_saurion.h, 72	
saurion, 37	on_wrote_arg, 50
Sauriori, or	saurion_create
S	LowSaurion, 18
Saurion, 43	saurion_destroy
saurion_wrapper, 51	LowSaurion, 19
s mtx	saurion_send
ThreadPool, 61	LowSaurion, 20
Saurion, 39	saurion_start
~Saurion, 40	LowSaurion, 20
	saurion_stop
ClosedCb, 39	LowSaurion, 21
ConnectedCb, 39	saurion_worker_master
ErrorCb, 40	low_saurion.c, 93
init, 41	saurion_worker_slave
on_closed, 41	low_saurion.c, 94
on_connected, 41	saurion_wrapper, 51
on_error, 42	s, 51
on_readed, 42	sel, 51
on_wrote, 42	sel
operator=, 42, 43	saurion_wrapper, 51
ReadedCb, 40	

send	ThreadPool, 55, 56
Saurion, 43	wait_closeable, 59
set_request	wait_empty, 59
LowSaurion, 21	threadpool.cpp
size	T, 107
Node, 33	TP, 107
SS	threadpool.hpp
low_saurion.h, 72	FORCE, 76
saurion, 38	KINDLY, 76
status	StopFlags, 76
low_saurion.h, 72	ThreadPool::AsyncMultiQueue, 25
saurion, 38	~AsyncMultiQueue, 26
status_c	AsyncMultiQueue, 25, 26
low_saurion.h, 73	clear, 26
saurion, 38	empty, 26
status_m low saurion.h, 73	front, 27
saurion, 38	m_it, 28 m_queues, 29
	new_queue, 27
stop Saurion, 43	operator=, 27
ThreadPool, 58	pop, 28
StopFlags	push, 28
threadpool.hpp, 76	remove_queue, 28
till Caapoolinpp, 70	ThreadPool::AsyncQueue, 29
T	~AsyncQueue, 30
threadpool.cpp, 107	AsyncQueue, 29, 30
Task, 51	empty, 30
$\sim$ Task, 52	front, 30
argument, 53	m cnt, 32
function, 53	m max, 32
operator=, 52, 53	m_mtx, 32
Task, 52	m_queue, 32
thread_entry	operator=, 31
ThreadPool, 58	pop, <b>31</b>
thread_worker	push, 31
ThreadPool, 59	ThreadPool add
ThreadPool, 53	cthreadpool.cpp, 78
$\sim$ ThreadPool, 56	cthreadpool.hpp, 64
add, 56	ThreadPool_add_default
AsyncMultiQueue, 55	cthreadpool.cpp, 78
AsyncQueue, 55	cthreadpool.hpp, 64
cthreadpool.hpp, 63	ThreadPool_create
empty, 57	cthreadpool.cpp, 79
init, 57	cthreadpool.hpp, 64
m_faccept, 60	ThreadPool_create_default
m_fstop, 60	cthreadpool.cpp, 79
m_nth, 60	cthreadpool.hpp, 64
m_q_cond, 60	ThreadPool_destroy
m_q_mtx, 60	cthreadpool.cpp, 79
m_queues, 61	cthreadpool.hpp, 64
m_started, 61	ThreadPool_empty
m_ths, 61	cthreadpool.cpp, 79
new_queue, 57 operator=, 57, 58	cthreadpool.hpp, 65
remove_queue, 58	ThreadPool_init
s_mtx, 61	cthreadpool.cpp, 79
stop, 58	cthreadpool.hpp, 65
thread_entry, 58	ThreadPool_new_queue
thread_worker, 59	cthreadpool.cpp, 80
undau_worker, Ja	

cthreadpool.hpp, 65 ThreadPool\_remove\_queue cthreadpool.cpp, 80 cthreadpool.hpp, 65 ThreadPool\_stop cthreadpool.cpp, 80 cthreadpool.hpp, 65 ThreadPool\_wait\_empty cthreadpool.cpp, 80 cthreadpool.hpp, 66 TP threadpool.cpp, 107 wait\_closeable ThreadPool, 59 wait\_empty ThreadPool, 59 WroteCb Saurion, 40