Saurion

Generated by Doxygen 1.9.4

Chapter 1

Todo List

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

add message contraint

validar `msg_size`, crear maximos

validar `offsets`

Chapter 2

Module Index

2.1 Modules

Here is a list of all modules:

Chapter 3

Class Index

3.1 Class List

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

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Module Documentation

5.1 LowSaurion

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

Classes

- struct saurion

  Main structure for managing io_uring and socket events.

Macros

- #define _POSIX_C_SOURCE 200809L
- #define PACKING_SZ 32

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

Functions

- int saurion_set_socket (int p)

  Creates a socket.
- struct saurion * saurion_create (uint32_t n_threads)

  Creates an instance of the `saurion` structure.
- int saurion_start (struct saurion *s)

  Starts event processing in the `saurion` structure.
- void saurion_stop (const struct saurion *s)

  Stops event processing in the `saurion` structure.
- void saurion_destroy (struct saurion *s)

  Destroys the `saurion` structure and frees all associated resources.
- void saurion_send (struct saurion *s, const int fd, const char *const msg)

  Sends a message through a socket using io_uring.
- int allocate_iovec (struct iovec *iov, size_t amount, size_t pos, size_t size, void **chd_ptr)
- int initialize_iovec (struct iovec *iov, size_t amount, size_t pos, const void *msg, size_t size, uint8_t h)

  Initializes a specified `iovec` structure with a message fragment.
- int set_request (struct request **r, struct Node **l, 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:

```
First iovec (8192 bytes):
+----------------------------------------+-----------------------+
| iov_base                               | iov_len               |
+----------------------------------------+-----------------------+
| 8 bytes header, 8184 bytes of message  | 8192                  |
+----------------------------------------+-----------------------+

Second iovec (817 bytes):
+----------------------------------------+-----------------------+
| iov_base                               | iov_len               |
+----------------------------------------+-----------------------+
| 816 bytes of message, 1 byte footer (0) | 817                  |
+----------------------------------------+-----------------------+
```

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:

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

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

Author

Israel

Date

2024

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

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

Parameters

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

Return values

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

Note

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

5.1.2 Macro Definition Documentation

5.1.2.1 _POSIX_C_SOURCE

```
#define _POSIX_C_SOURCE 200809L
```

Definition at line 107 of file low_saurion.h.

5.1.2.2 PACKING_SZ

```
#define PACKING_SZ 32
```

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

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

In this case, the value is set to 32 bytes, meaning that structures marked with `__attribute__((aligned(PACKING_SZ)))` will be aligned to 32-byte boundaries.

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

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

Definition at line 139 of file low_saurion.h.

5.1.3 Function Documentation

5.1.3.1 allocate_iovec()

```
int allocate_iovec (
              struct iovec * iov,
              size_t amount,
              size_t pos,
              size_t size,
              void ** chd_ptr )
```

Definition at line 164 of file low_saurion.c.

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

5.1.3.2 free_request()

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

Definition at line 94 of file low_saurion.c.

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

5.1.3.3 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 115 of file low_saurion.c.

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

```
00134               dest += sizeof (uint64_t);
00135               len -= sizeof (uint64_t);
00136             }
00137           else
00138             {
00139               orig -= sizeof (uint64_t);
00140             }
00141           if ((pos + 1) == amount)
00142             {
00143               --len;
00144               cpy_sz = (len < size ?  len :  size);
00145               dest[cpy_sz] = 0;
00146             }
00147         }
00148       cpy_sz = (len < size ?  len :  size);
00149       memcpy (dest, orig, cpy_sz);
00150       dest += cpy_sz;
00151       size_t rem = CHUNK_SZ - (dest - (char *)iov->iov_base);
00152       memset (dest, 0, rem);
00153     }
00154   else
00155     {
00156       memset ((char *)iov->iov_base, 0, CHUNK_SZ);
00157     }
00158   return SUCCESS_CODE;
00159 }
```

5.1.3.4 read_chunk()

```
int read_chunk (
              void ** dest,
              size_t * len,
              struct request *const req )  [private]
```

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 iovec buffers and state information. The function updates the request's state to track the current position within the iovecs and any incomplete messages.

Note

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

Warning

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

Returns

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

Return values

SUCCESS_CODE	No malformed message found.
ERROR_CODE	Malformed message found.

Todo add message contraint

validar `msg_size`, crear maximos

validar `offsets`

Definition at line 688 of file low_saurion.c.

```
00689 {
00690   struct chunk_params p;
00691   p.req = req;
00692   p.dest = dest;
00693   p.len = len;
00694   if (p.req->iovec_count == 0)
00695     {
00696       return ERROR_CODE;
00697     }
00698
00699   p.max_iov_cont = calculate_max_iov_content (p.req);
00700   p.cont_sz = 0;
00701   p.cont_rem = 0;
00702   p.curr_iov = 0;
00703   p.curr_iov_off = 0;
00704   p.dest_off = 0;
00705   p.dest_ptr = NULL;
00706   if (!prepare_destination (&p))
00707     {
00708       return ERROR_CODE;
00709     }
00710
00711   uint8_t ok = 1UL;
00712   copy_data (&p, &ok);
00713
00714   if (validate_and_update (&p, ok))
00715     {
00716       return SUCCESS_CODE;
00717     }
00718   read_chunk_free (&p);
00719   return ERROR_CODE;
00720 }
```

5.1.3.5 saurion_create()

```
struct saurion * saurion_create (
            uint32_t n_threads )
```

Creates an instance of the `saurion` structure.

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

Parameters

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

Returns

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

Definition at line 835 of file low_saurion.c.

```
00836 {
00837   LOG_INIT (" ");
00838   struct saurion *p = (struct saurion *)malloc (sizeof (struct saurion));
00839   if (!p)
00840     {
00841       LOG_END (" ");
00842       return NULL;
00843     }
00844   int ret = 0;
00845   ret = pthread_mutex_init (&p->status_m, NULL);
00846   if (ret)
00847     {
00848       free (p);
00849       LOG_END (" ");
00850       return NULL;
00851     }
00852   ret = pthread_cond_init (&p->status_c, NULL);
00853   if (ret)
00854     {
00855       free (p);
00856       LOG_END (" ");
00857       return NULL;
00858     }
00859   p->m_rings
00860       = (pthread_mutex_t *)malloc (n_threads * sizeof (pthread_mutex_t));
00861   if (!p->m_rings)
00862     {
00863       free (p);
00864       LOG_END (" ");
00865       return NULL;
00866     }
00867   for (uint32_t i = 0; i < n_threads; ++i)
00868     {
00869       pthread_mutex_init (&(p->m_rings[i]), NULL);
00870     }
00871   p->ss = 0;
00872   n_threads = (n_threads < 2 ?  2 :  n_threads);
00873   n_threads = (n_threads > NUM_CORES ? NUM_CORES : n_threads);
00874   p->n_threads = n_threads;
00875   p->status = 0;
00876   p->list = NULL;
00877   p->cb.on_connected = NULL;
00878   p->cb.on_connected_arg = NULL;
00879   p->cb.on_readed = NULL;
00880   p->cb.on_readed_arg = NULL;
00881   p->cb.on_wrote = NULL;
00882   p->cb.on_wrote_arg = NULL;
00883   p->cb.on_closed = NULL;
00884   p->cb.on_closed_arg = NULL;
00885   p->cb.on_error = NULL;
00886   p->cb.on_error_arg = NULL;
00887   p->next = 0;
00888   p->efds = (int *)malloc (sizeof (int) * p->n_threads);
00889   if (!p->efds)
00890     {
00891       free (p->m_rings);
00892       free (p);
00893       LOG_END (" ");
00894       return NULL;
00895     }
00896   for (uint32_t i = 0; i < p->n_threads; ++i)
00897     {
00898       p->efds[i] = eventfd (0, EFD_NONBLOCK);
00899       if (p->efds[i] == ERROR_CODE)
00900         {
00901           for (uint32_t j = 0; j < i; ++j)
00902             {
00903               close (p->efds[j]);
00904             }
00905           free (p->efds);
00906           free (p->m_rings);
```

```
00907           free (p);
00908           LOG_END (" ");
00909           return NULL;
00910         }
00911     }
00912   p->rings
00913       = (struct io_uring *)malloc (sizeof (struct io_uring) * p->n_threads);
00914   if (!p->rings)
00915     {
00916       for (uint32_t j = 0; j < p->n_threads; ++j)
00917         {
00918           close (p->efds[j]);
00919         }
00920       free (p->efds);
00921       free (p->m_rings);
00922       free (p);
00923       LOG_END (" ");
00924       return NULL;
00925     }
00926   for (uint32_t i = 0; i < p->n_threads; ++i)
00927     {
00928       memset (&p->rings[i], 0, sizeof (struct io_uring));
00929       ret = io_uring_queue_init (SAURION_RING_SIZE, &p->rings[i], 0);
00930       if (ret)
00931         {
00932           for (uint32_t j = 0; j < p->n_threads; ++j)
00933             {
00934               close (p->efds[j]);
00935             }
00936           free (p->efds);
00937           free (p->rings);
00938           free (p->m_rings);
00939           free (p);
00940           LOG_END (" ");
00941           return NULL;
00942         }
00943     }
00944   p->pool = threadpool_create (p->n_threads);
00945   LOG_END (" ");
00946   return p;
00947 }
```

5.1.3.6 saurion_destroy()

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

Destroys the `saurion` structure and frees all associated resources.

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

Parameters

s	Pointer to the `saurion` structure.

Definition at line 1189 of file low_saurion.c.

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

```
01204   list_free (&s->list);
01205   for (uint32_t i = 0; i < s->n_threads; ++i)
01206     {
01207       close (s->efds[i]);
01208     }
01209   free (s->efds);
01210   if (!s->ss)
01211     {
01212       close (s->ss);
01213     }
01214   free (s->rings);
01215   pthread_mutex_destroy (&s->status_m);
01216   pthread_cond_destroy (&s->status_c);
01217   free (s);
01218 }
```

5.1.3.7 saurion_send()

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

Sends a message through a socket using io_uring.

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

Parameters

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

Definition at line 1222 of file low_saurion.c.

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

5.1.3.8 saurion_set_socket()

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

Creates a socket.

Creates and sets a socket, ready for saurion configuration.

Parameters

p	port

Returns

result of socket creation.

Definition at line 797 of file low_saurion.c.

```
00798 {
00799   int sock = 0;
00800   struct sockaddr_in srv_addr;
00801
00802   sock = socket (PF_INET, SOCK_STREAM, 0);
00803   if (sock < 1)
00804     {
00805       return ERROR_CODE;
00806     }
00807
00808   int enable = 1;
00809   if (setsockopt (sock, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof (int)) < 0)
00810     {
00811       return ERROR_CODE;
00812     }
00813
00814   memset (&srv_addr, 0, sizeof (srv_addr));
00815   srv_addr.sin_family = AF_INET;
00816   srv_addr.sin_port = htons (p);
00817   srv_addr.sin_addr.s_addr = htonl (INADDR_ANY);
00818
00819   if (bind (sock, (const struct sockaddr *)&srv_addr, sizeof (srv_addr)) < 0)
00820     {
00821       return ERROR_CODE;
00822     }
00823
00824   if (listen (sock, ACCEPT_QUEUE) < 0)
00825     {
00826       return ERROR_CODE;
00827     }
00828
00829   return sock;
00830 }
```

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 1147 of file low_saurion.c.

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

```
01156         {
01157           return ERROR_CODE;
01158         }
01159       ss->s = s;
01160       ss->sel = i;
01161       threadpool_add (s->pool, saurion_worker_slave, ss);
01162     }
01163   pthread_mutex_lock (&s->status_m);
01164   while (s->status < (int)s->n_threads)
01165     {
01166       pthread_cond_wait (&s->status_c, &s->status_m);
01167     }
01168   pthread_mutex_unlock (&s->status_m);
01169   return SUCCESS_CODE;
01170 }
```

5.1.3.10 saurion_stop()

```
void saurion_stop (
            const struct saurion * s )
```

Stops event processing in the `saurion` structure.

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

Parameters

s	Pointer to the `saurion` structure.

Definition at line 1174 of file low_saurion.c.

```
01175 {
01176   uint64_t u = 1;
01177   for (uint32_t i = 0; i < s->n_threads; ++i)
01178     {
01179       while (write (s->efds[i], &u, sizeof (u)) < 0)
01180         {
01181           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
01182         }
01183     }
01184   threadpool_wait_empty (s->pool);
01185 }
```

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

Returns

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

Return values

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

Note

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

Definition at line 188 of file low_saurion.c.

```
00190 {
00191   uint64_t full_size = s;
00192   if (h)
00193     {
00194       full_size += (sizeof (uint64_t) + sizeof (uint8_t));
00195     }
00196   size_t amount = full_size / CHUNK_SZ;
00197   amount = amount + (full_size % CHUNK_SZ == 0 ?  0 :  1);
00198   struct request *temp = (struct request *)malloc (
00199       sizeof (struct request) + sizeof (struct iovec) * amount);
00200   if (!temp)
00201     {
00202       return ERROR_CODE;
00203     }
00204   if (!*r)
00205     {
00206       *r = temp;
00207       (*r)->prev = NULL;
00208       (*r)->prev_size = 0;
00209       (*r)->prev_remain = 0;
00210       (*r)->next_iov = 0;
00211       (*r)->next_offset = 0;
00212     }
00213   else
00214     {
00215       temp->client_socket = (*r)->client_socket;
00216       temp->event_type = (*r)->event_type;
00217       temp->prev = (*r)->prev;
00218       temp->prev_size = (*r)->prev_size;
00219       temp->prev_remain = (*r)->prev_remain;
00220       temp->next_iov = (*r)->next_iov;
00221       temp->next_offset = (*r)->next_offset;
00222       *r = temp;
00223     }
00224   struct request *req = *r;
00225   req->iovec_count = (int)amount;
00226   void **children_ptr = (void **)malloc (amount * sizeof (void *));
00227   if (!children_ptr)
00228     {
00229       free_request (req, children_ptr, 0);
00230       return ERROR_CODE;
00231     }
```

```
00232   for (size_t i = 0; i < amount; ++i)
00233     {
00234       if (!allocate_iovec (&req->iov[i], amount, i, full_size, children_ptr))
00235         {
00236           free_request (req, children_ptr, amount);
00237           return ERROR_CODE;
00238         }
00239       if (!initialize_iovec (&req->iov[i], amount, i, m, s, h))
00240         {
00241           free_request (req, children_ptr, amount);
00242           return ERROR_CODE;
00243         }
00244     }
00245   if (list_insert (l, req, amount, children_ptr))
00246     {
00247       free_request (req, children_ptr, amount);
00248       return ERROR_CODE;
00249     }
00250   free (children_ptr);
00251   return SUCCESS_CODE;
00252 }
```

5.2 ThreadPool

Functions

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

5.2.1 Detailed Description

5.2.2 Function Documentation

5.2.2.1 threadpool_add()

```
void threadpool_add (
            struct threadpool * pool,
            void(*)(void *) function,
            void * argument )
```

Definition at line 175 of file threadpool.c.

```
00177 {
00178   LOG_INIT (" ");
00179   if (pool == NULL || function == NULL)
00180     {
00181       LOG_END (" ");
00182       return;
00183     }
00184
00185   struct task *new_task = malloc (sizeof (struct task));
00186   if (new_task == NULL)
00187     {
00188       perror ("Failed to allocate task");
00189       LOG_END (" ");
```

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

5.2.2.2 threadpool_create()

```
struct threadpool * threadpool_create (
            size_t num_threads )
```

Definition at line 32 of file threadpool.c.

```
00033 {
00034   LOG_INIT (" ");
00035   struct threadpool *pool = malloc (sizeof (struct threadpool));
00036   if (pool == NULL)
00037     {
00038       perror ("Failed to allocate threadpool");
00039       LOG_END (" ");
00040       return NULL;
00041     }
00042   if (num_threads < 3)
00043     {
00044       num_threads = 3;
00045     }
00046   if (num_threads > NUM_CORES)
00047     {
00048       num_threads = NUM_CORES;
00049     }
00050
00051   pool->num_threads = num_threads;
00052   pool->threads = malloc (sizeof (pthread_t) * num_threads);
00053   if (pool->threads == NULL)
00054     {
00055       perror ("Failed to allocate threads array");
00056       free (pool);
00057       LOG_END (" ");
00058       return NULL;
00059     }
00060
00061   pool->task_queue_head = NULL;
00062   pool->task_queue_tail = NULL;
00063   pool->stop = FALSE;
00064   pool->started = FALSE;
00065
00066   if (pthread_mutex_init (&pool->queue_lock, NULL) != 0)
00067     {
00068       perror ("Failed to initialize mutex");
00069       free (pool->threads);
00070       free (pool);
00071       LOG_END (" ");
00072       return NULL;
00073     }
00074
00075   if (pthread_cond_init (&pool->queue_cond, NULL) != 0)
00076     {
00077       perror ("Failed to initialize condition variable");
00078       pthread_mutex_destroy (&pool->queue_lock);
00079       free (pool->threads);
```

```
00080       free (pool);
00081       LOG_END (" ");
00082       return NULL;
00083     }
00084
00085   if (pthread_cond_init (&pool->empty_cond, NULL) != 0)
00086     {
00087       perror ("Failed to initialize empty condition variable");
00088       pthread_mutex_destroy (&pool->queue_lock);
00089       pthread_cond_destroy (&pool->queue_cond);
00090       free (pool->threads);
00091       free (pool);
00092       LOG_END (" ");
00093       return NULL;
00094     }
00095
00096   LOG_END (" ");
00097   return pool;
00098 }
```

5.2.2.3 threadpool_create_default()

```
struct threadpool * threadpool_create_default (
              void  )
```

Definition at line 101 of file threadpool.c.

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

5.2.2.4 threadpool_destroy()

```
void threadpool_destroy (
              struct threadpool * pool )
```

Definition at line 274 of file threadpool.c.

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

5.2.2.5 threadpool_empty()

```
int threadpool_empty (
            struct threadpool * pool )
```

Definition at line 240 of file threadpool.c.

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

5.2.2.6 threadpool_init()

```
void threadpool_init (
            struct threadpool * pool )
```

Definition at line 151 of file threadpool.c.

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

5.2.2.7 threadpool_stop()

```
void threadpool_stop (
            struct threadpool * pool )
```

Definition at line 216 of file threadpool.c.

```
00217 {
00218   LOG_INIT (" ");
00219   if (pool == NULL || !pool->started)
00220     {
00221       LOG_END (" ");
00222       return;
00223     }
00224   threadpool_wait_empty (pool);
00225
00226   pthread_mutex_lock (&pool->queue_lock);
00227   pool->stop = TRUE;
00228   pthread_cond_broadcast (&pool->queue_cond);
```

```
00229   pthread_mutex_unlock (&pool->queue_lock);
00230
00231   for (size_t i = 0; i < pool->num_threads; i++)
00232     {
00233       pthread_join (pool->threads[i], NULL);
00234     }
00235   pool->started = FALSE;
00236   LOG_END (" ");
00237 }
```

5.2.2.8 threadpool_wait_empty()

```
void threadpool_wait_empty (
             struct threadpool * pool )
```

Definition at line 256 of file threadpool.c.

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

Chapter 6

Class Documentation

6.1 chunk_params Struct Reference

Collaboration diagram for chunk_params:

Public Attributes

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

6.1.1 Detailed Description

Definition at line 452 of file low_saurion.c.

6.1.2 Member Data Documentation

6.1.2.1 cont_rem

```
size_t chunk_params::cont_rem
```

Definition at line 459 of file low_saurion.c.

6.1.2.2 cont_sz

```
size_t chunk_params::cont_sz
```

Definition at line 458 of file low_saurion.c.

6.1.2.3 curr_iov

```
size_t chunk_params::curr_iov
```

Definition at line 461 of file low_saurion.c.

6.1.2.4 curr_iov_off

```
size_t chunk_params::curr_iov_off
```

Definition at line 462 of file low_saurion.c.

6.1.2.5 dest

```
void** chunk_params::dest
```

Definition at line 454 of file low_saurion.c.

6.1.2.6 dest_off

```
size_t chunk_params::dest_off
```

Definition at line 456 of file low_saurion.c.

6.1.2.7 dest_ptr

```
void* chunk_params::dest_ptr
```

Definition at line 455 of file low_saurion.c.

6.1.2.8 len

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

Definition at line 463 of file low_saurion.c.

6.1.2.9 max_iov_cont

```
size_t chunk_params::max_iov_cont
```

Definition at line 460 of file low_saurion.c.

6.1.2.10 req

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

Definition at line 457 of file low_saurion.c.

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

- /__w/saurion/saurion/src/low_saurion.c

6.2 Node Struct Reference

Collaboration diagram for Node:

Public Attributes

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

6.2.1 Detailed Description

Definition at line 6 of file linked_list.c.

6.2.2 Member Data Documentation

6.2.2.1 children

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

Definition at line 10 of file linked_list.c.

6.2.2.2 next

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

Definition at line 11 of file linked_list.c.

6.2.2.3 ptr

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

Definition at line 8 of file linked_list.c.

6.2.2.4 size

```
size_t Node::size
```

Definition at line 9 of file linked_list.c.

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

- /__w/saurion/saurion/src/linked_list.c

6.3 request Struct Reference

Public Attributes

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

6.3.1 Detailed Description

Definition at line 32 of file low_saurion.c.

6.3.2 Member Data Documentation

6.3.2.1 client_socket

```
int request::client_socket
```

Definition at line 41 of file low_saurion.c.

6.3.2.2 event_type

```
int request::event_type
```

Definition at line 39 of file low_saurion.c.

6.3.2.3 iov

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

Definition at line 42 of file low_saurion.c.

6.3.2.4 iovec_count

```
size_t request::iovec_count
```

Definition at line 40 of file low_saurion.c.

6.3.2.5 next_iov

```
size_t request::next_iov
```

Definition at line 37 of file low_saurion.c.

6.3.2.6 next_offset

```
size_t request::next_offset
```

Definition at line 38 of file low_saurion.c.

6.3.2.7 prev

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

Definition at line 34 of file low_saurion.c.

6.3.2.8 prev_remain

```
size_t request::prev_remain
```

Definition at line 36 of file low_saurion.c.

6.3.2.9 prev_size

```
size_t request::prev_size
```

Definition at line 35 of file low_saurion.c.

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

- /__w/saurion/saurion/src/low_saurion.c

6.4 saurion Struct Reference

Main structure for managing io_uring and socket events.

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

Collaboration diagram for saurion:

Classes

- struct saurion_callbacks

  Structure containing callback functions to handle socket events.

Public Attributes

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

6.4.1 Detailed Description

Main structure for managing io_uring and socket events.

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

Definition at line 148 of file low_saurion.h.

6.4.2 Member Data Documentation

6.4.2.1 efds

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

Eventfd descriptors used for internal signaling between threads.

Definition at line 157 of file low_saurion.h.

6.4.2.2 list

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

Linked list for storing active requests.

Definition at line 159 of file low_saurion.h.

6.4.2.3 m_rings

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

Array of mutexes to protect the io_uring rings.

Definition at line 153 of file low_saurion.h.

6.4.2.4 n_threads

```
uint32_t saurion::n_threads
```

Number of threads in the thread pool.

Definition at line 169 of file low_saurion.h.

6.4.2.5 next

```
uint32_t saurion::next
```

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

Definition at line 171 of file low_saurion.h.

6.4.2.6 pool

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

Thread pool for executing tasks in parallel.

Definition at line 167 of file low_saurion.h.

6.4.2.7 rings

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

Array of io_uring structures for managing the event queue.

Definition at line 151 of file low_saurion.h.

6.4.2.8 ss

```
int saurion::ss
```

Server socket descriptor for accepting connections.

Definition at line 155 of file low_saurion.h.

6.4.2.9 status

```
int saurion::status
```

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

Definition at line 165 of file low_saurion.h.

6.4.2.10 status_c

```
pthread_cond_t saurion::status_c
```

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

Definition at line 163 of file low_saurion.h.

6.4.2.11 status_m

```
pthread_mutex_t saurion::status_m
```

Mutex to protect the state of the structure.

Definition at line 161 of file low_saurion.h.

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

- /__w/saurion/saurion/include/low_saurion.h

6.5 Saurion Class Reference

```
#include <saurion.hpp>
```

Collaboration diagram for Saurion:

Public Types

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

Public Member Functions

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

Private Attributes

- struct saurion ∗ s

6.5.1 Detailed Description

Definition at line 7 of file saurion.hpp.

6.5.2 Member Typedef Documentation

6.5.2.1 ClosedCb

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

Definition at line 14 of file saurion.hpp.

6.5.2.2 ConnectedCb

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

Definition at line 10 of file saurion.hpp.

6.5.2.3 ErrorCb

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

Definition at line 15 of file saurion.hpp.

6.5.2.4 ReadedCb

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

Definition at line 11 of file saurion.hpp.

6.5.2.5 WroteCb

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

Definition at line 13 of file saurion.hpp.

6.5.3 Constructor & Destructor Documentation

6.5.3.1 Saurion() [1/3]

```
Saurion::Saurion (
            const uint32_t thds,
            const int sck )  [explicit], [noexcept]
```

Definition at line 5 of file saurion.cpp.

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

6.5.3.2 ~Saurion()

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

Definition at line 15 of file saurion.cpp.

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

6.5.3.3 Saurion() [2/3]

```
Saurion::Saurion (
            const Saurion &  )  [delete]
```

6.5.3.4 Saurion() [3/3]

```
Saurion::Saurion (
            Saurion &&  )  [delete]
```

6.5.4 Member Function Documentation

6.5.4.1 init()

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

Definition at line 18 of file saurion.cpp.

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

6.5.4.2 on_closed()

```
Saurion * Saurion::on_closed (
            Saurion::ClosedCb ncb,
            void * arg )  [noexcept]
```

Definition at line 57 of file saurion.cpp.

```
00058 {
00059   s->cb.on_closed = ncb;
00060   s->cb.on_closed_arg = arg;
00061   return this;
00062 }
```

6.5.4.3 on_connected()

```
Saurion * Saurion::on_connected (
            Saurion::ConnectedCb ncb,
            void * arg )  [noexcept]
```

Definition at line 33 of file saurion.cpp.

```
00034 {
00035   s->cb.on_connected = ncb;
00036   s->cb.on_connected_arg = arg;
00037   return this;
00038 }
```

6.5.4.4 on_error()

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

Definition at line 65 of file saurion.cpp.

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

6.5.4.5 on_readed()

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

Definition at line 41 of file saurion.cpp.

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

6.5.4.6 on_wrote()

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

Definition at line 49 of file saurion.cpp.

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

6.5.4.7 operator=() [1/2]

```
Saurion & Saurion::operator= (
            const Saurion &  )  [delete]
```

6.5.4.8 operator=() [2/2]

```
Saurion & Saurion::operator= (
            Saurion &&  )  [delete]
```

6.5.4.9 send()

```
void Saurion::send (
            const int fd,
            const char *const msg )  [noexcept]
```

Definition at line 73 of file saurion.cpp.

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

6.5.4.10 stop()

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

Definition at line 27 of file saurion.cpp.

```
00028 {
00029   saurion_stop (this->s);
00030 }
```

6.5.5 Member Data Documentation

6.5.5.1 s

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

Definition at line 38 of file saurion.hpp.

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

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

6.6 saurion::saurion_callbacks Struct Reference

Structure containing callback functions to handle socket events.

`#include <low_saurion.h>`

Public Attributes

- void(∗ on_connected)(const int fd, void ∗arg)

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

  Callback for handling read events.
- void ∗ on_readed_arg
- void(∗ on_wrote)(const int fd, void ∗arg)

  Callback for handling write events.
- void ∗ on_wrote_arg
- void(∗ on_closed)(const int fd, void ∗arg)

  Callback for handling socket closures.
- void ∗ on_closed_arg
- void(∗ on_error)(const int fd, const char ∗const content, const ssize_t len, void ∗arg)

  Callback for handling error events.
- void ∗ on_error_arg

6.6.1 Detailed Description

Structure containing callback functions to handle socket events.

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

Definition at line 181 of file low_saurion.h.

6.6.2 Member Data Documentation

6.6.2.1 on_closed

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

Callback for handling socket closures.

Parameters

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

Definition at line 221 of file low_saurion.h.

6.6.2.2 on_closed_arg

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

Additional argument for the close callback.

Definition at line 223 of file low_saurion.h.

6.6.2.3 on_connected

```
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 189 of file low_saurion.h.

6.6.2.4 on_connected_arg

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

Additional argument for the connection callback.

Definition at line 191 of file low_saurion.h.

6.6.2.5 on_error

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

Callback for handling error events.

Parameters

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

Definition at line 233 of file low_saurion.h.

6.6.2.6 on_error_arg

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

Additional argument for the error callback.

Definition at line 236 of file low_saurion.h.

6.6.2.7 on_readed

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

Callback for handling read events.

Parameters

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

Definition at line 201 of file low_saurion.h.

6.6.2.8 on_readed_arg

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

Additional argument for the read callback.

Definition at line 204 of file low_saurion.h.

6.6.2.9 on_wrote

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

Callback for handling write events.

Parameters

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

Definition at line 212 of file low_saurion.h.

6.6.2.10 on_wrote_arg

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

Additional argument for the write callback.

Definition at line 213 of file low_saurion.h.

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

- /__w/saurion/saurion/include/low_saurion.h

6.7 saurion_callbacks Struct Reference

Structure containing callback functions to handle socket events.

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

Public Attributes

- void(∗ on_connected)(const int fd, void ∗arg)

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

  Callback for handling read events.
- void ∗ on_readed_arg
- void(∗ on_wrote)(const int fd, void ∗arg)

  Callback for handling write events.
- void ∗ on_wrote_arg
- void(∗ on_closed)(const int fd, void ∗arg)

  Callback for handling socket closures.
- void ∗ on_closed_arg
- void(∗ on_error)(const int fd, const char ∗const content, const ssize_t len, void ∗arg)

  Callback for handling error events.
- void ∗ on_error_arg

6.7.1 Detailed Description

Structure containing callback functions to handle socket events.

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

Definition at line 31 of file low_saurion.h.

6.7.2 Member Data Documentation

6.7.2.1 on_closed

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

Callback for handling socket closures.

Parameters

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

Definition at line 71 of file low_saurion.h.

6.7.2.2 on_closed_arg

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

Additional argument for the close callback.

Definition at line 73 of file low_saurion.h.

6.7.2.3 on_connected

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

Callback for handling new connections.

Parameters

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

Definition at line 39 of file low_saurion.h.

6.7.2.4 on_connected_arg

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

Additional argument for the connection callback.

Definition at line 41 of file low_saurion.h.

6.7.2.5 on_error

```
void(* saurion_callbacks::on_error) (const int fd, const char *const content, const ssize_↩
t len, void *arg)
```

Callback for handling error events.

Parameters

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

Definition at line 83 of file low_saurion.h.

6.7.2.6 on_error_arg

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

Additional argument for the error callback.

Definition at line 86 of file low_saurion.h.

6.7.2.7 on_readed

```
void(* 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 51 of file low_saurion.h.

6.7.2.8 on_readed_arg

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

Additional argument for the read callback.

Definition at line 54 of file low_saurion.h.

6.7.2.9 on_wrote

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

Callback for handling write events.

Parameters

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

Definition at line 62 of file low_saurion.h.

6.7.2.10 on_wrote_arg

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

Additional argument for the write callback.

Definition at line 63 of file low_saurion.h.

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

- /__w/saurion/saurion/include/low_saurion.h

6.8 saurion_wrapper Struct Reference

Collaboration diagram for saurion_wrapper:

Public Attributes

- struct saurion * s
- uint32_t sel

6.8.1 Detailed Description

Definition at line 50 of file low_saurion.c.

6.8.2 Member Data Documentation

6.8.2.1 s

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

Definition at line 52 of file low_saurion.c.

6.8.2.2 sel

```
uint32_t saurion_wrapper::sel
```

Definition at line 53 of file low_saurion.c.

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

- /__w/saurion/saurion/src/low_saurion.c

6.9 task Struct Reference

Collaboration diagram for task:

Public Attributes

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

6.9.1 Detailed Description

Definition at line 11 of file threadpool.c.

6.9.2 Member Data Documentation

6.9.2.1 argument

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

Definition at line 14 of file threadpool.c.

6.9.2.2 function

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

Definition at line 13 of file threadpool.c.

6.9.2.3 next

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

Definition at line 15 of file threadpool.c.

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

- /__w/saurion/saurion/src/threadpool.c

6.10 threadpool Struct Reference

Collaboration diagram for threadpool:

Public Attributes

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

6.10.1 Detailed Description

Definition at line 18 of file threadpool.c.

6.10.2 Member Data Documentation

6.10.2.1 empty_cond

```
pthread_cond_t threadpool::empty_cond
```

Definition at line 26 of file threadpool.c.

6.10.2.2 num_threads

```
size_t threadpool::num_threads
```

Definition at line 21 of file threadpool.c.

6.10.2.3 queue_cond

```
pthread_cond_t threadpool::queue_cond
```

Definition at line 25 of file threadpool.c.

6.10.2.4 queue_lock

```
pthread_mutex_t threadpool::queue_lock
```

Definition at line 24 of file threadpool.c.

6.10.2.5 started

```
int threadpool::started
```

Definition at line 28 of file threadpool.c.

6.10.2.6 stop

```
int threadpool::stop
```

Definition at line 27 of file threadpool.c.

6.10.2.7 task_queue_head

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

Definition at line 22 of file threadpool.c.

6.10.2.8 task_queue_tail

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

Definition at line 23 of file threadpool.c.

6.10.2.9 threads

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

Definition at line 20 of file threadpool.c.

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

- /__w/saurion/saurion/src/threadpool.c

Chapter 7

File Documentation

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

`#include <stddef.h>`
Include dependency graph for linked_list.h: This graph shows which files directly or indirectly include this file:

Functions

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

7.1.1 Function Documentation

7.1.1.1 list_delete_node()

```
void list_delete_node (
            struct Node ** head,
            const void *const ptr )
```

Definition at line 106 of file linked_list.c.

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

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

7.1.1.2 list_free()

```
void list_free (
              struct Node ** head )
```

Definition at line 138 of file linked_list.c.

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

7.1.1.3 list_insert()

```
int list_insert (
              struct Node ** head,
              void * ptr,
              size_t amount,
              void ** children )
```

Definition at line 65 of file linked_list.c.

```
00066 {
00067   struct Node *new_node = create_node (ptr, amount, children);
00068   if (!new_node)
00069     {
00070       return 1;
00071     }
00072   pthread_mutex_lock (&list_mutex);
00073   if (!*head)
00074     {
00075       *head = new_node;
00076       pthread_mutex_unlock (&list_mutex);
00077       return 0;
00078     }
00079   struct Node *temp = *head;
00080   while (temp->next)
00081     {
00082       temp = temp->next;
00083     }
00084   temp->next = new_node;
00085   pthread_mutex_unlock (&list_mutex);
00086   return 0;
00087 }
```

7.2 linked_list.h

Go to the documentation of this file.

```
00001 #ifndef LINKED_LIST_H
00002 #define LINKED_LIST_H
00003
00004 #ifdef __cplusplus
00005 extern "C"
00006 {
00007 #endif
00008
00009 #include <stddef.h>
00010
00011   struct Node;
00012
00013   int list_insert (struct Node **head, void *ptr, size_t amount,
00014                    void **children);
00015
00016   void list_delete_node (struct Node **head, const void *const ptr);
00017
00018   void list_free (struct Node **head);
00019
00020 #ifdef __cplusplus
00021 }
00022 #endif
00023
00024 #endif // !LINKED_LIST_H
```

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

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

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

Classes

- struct saurion

  Main structure for managing io_uring and socket events.
- struct saurion::saurion_callbacks

  Structure containing callback functions to handle socket events.
- struct saurion_callbacks

  Structure containing callback functions to handle socket events.

Macros

- #define _POSIX_C_SOURCE 200809L
- #define PACKING_SZ 32

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

Functions

- int saurion_set_socket (int p)

  Creates a socket.
- struct saurion * saurion_create (uint32_t n_threads)

  Creates an instance of the `saurion` structure.
- int saurion_start (struct saurion *s)

  Starts event processing in the `saurion` structure.
- void saurion_stop (const struct saurion *s)

  Stops event processing in the `saurion` structure.
- void saurion_destroy (struct saurion *s)

  Destroys the `saurion` structure and frees all associated resources.
- void saurion_send (struct saurion *s, const int fd, const char *const msg)

  Sends a message through a socket using io_uring.

Variables

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

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

  Callback for handling read events.
- void * on_readed_arg
- void(* on_wrote)(const int fd, void *arg)

  Callback for handling write events.
- void * on_wrote_arg
- void(* on_closed)(const int fd, void *arg)

  Callback for handling socket closures.
- void * on_closed_arg
- void(* on_error)(const int fd, const char *const content, const ssize_t len, void *arg)

  Callback for handling error events.
- void * on_error_arg
- struct io_uring * rings
- pthread_mutex_t * m_rings
- int ss
- int * efds
- struct Node * list
- pthread_mutex_t status_m
- pthread_cond_t status_c
- int status
- struct threadpool * pool
- uint32_t n_threads
- uint32_t next

7.3.1 Variable Documentation

7.3.1.1 efds

```
int* efds
```

Eventfd descriptors used for internal signaling between threads.

Definition at line 7 of file low_saurion.h.

7.3.1.2 list

```
struct Node* list
```

Linked list for storing active requests.

Definition at line 9 of file low_saurion.h.

7.3.1.3 m_rings

```
pthread_mutex_t* m_rings
```

Array of mutexes to protect the io_uring rings.

Definition at line 3 of file low_saurion.h.

7.3.1.4 n_threads

```
uint32_t n_threads
```

Number of threads in the thread pool.

Definition at line 19 of file low_saurion.h.

7.3.1.5 next

```
uint32_t next
```

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

Definition at line 21 of file low_saurion.h.

7.3.1.6 on_closed

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

Callback for handling socket closures.

Parameters

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

Definition at line 38 of file low_saurion.h.

7.3.1.7 on_closed_arg

```
void * on_closed_arg
```

Additional argument for the close callback.

Definition at line 40 of file low_saurion.h.

7.3.1.8 on_connected

```
void(* on_connected)(const int fd, void *arg) (
            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 6 of file low_saurion.h.

7.3.1.9 on_connected_arg

```
void * on_connected_arg
```

Additional argument for the connection callback.

Definition at line 8 of file low_saurion.h.

7.3.1.10 on_error

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

Callback for handling error events.

Parameters

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

Definition at line 50 of file low_saurion.h.

7.3.1.11 on_error_arg

```
void * on_error_arg
```

Additional argument for the error callback.

Definition at line 53 of file low_saurion.h.

7.3.1.12 on_readed

```
void(* on_readed)(const int fd, const void *const content, const ssize_t len, void *arg) (
            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 18 of file low_saurion.h.

7.3.1.13 on_readed_arg

```
void * on_readed_arg
```

Additional argument for the read callback.

Definition at line 21 of file low_saurion.h.

7.3.1.14 on_wrote

```
void(* on_wrote)(const int fd, void *arg) (
            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 29 of file low_saurion.h.

7.3.1.15 on_wrote_arg

```
void * on_wrote_arg
```

Additional argument for the write callback.

Definition at line 30 of file low_saurion.h.

7.3.1.16 pool

```
struct threadpool* pool
```

Thread pool for executing tasks in parallel.

Definition at line 17 of file low_saurion.h.

7.3.1.17 rings

```
struct io_uring* rings
```

Array of io_uring structures for managing the event queue.

Definition at line 1 of file low_saurion.h.

7.3.1.18 ss

```
int ss
```

Server socket descriptor for accepting connections.

Definition at line 5 of file low_saurion.h.

7.3.1.19 status

```
int status
```

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

Definition at line 15 of file low_saurion.h.

7.3.1.20 status_c

```
pthread_cond_t status_c
```

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

Definition at line 13 of file low_saurion.h.

7.3.1.21 status_m

```
pthread_mutex_t status_m
```

Mutex to protect the state of the structure.

Definition at line 11 of file low_saurion.h.

7.4 low_saurion.h

Go to the documentation of this file.

```
00001
00104 #ifndef LOW_SAURION_H
00105 #define LOW_SAURION_H
00106
00107 #define _POSIX_C_SOURCE 200809L
00108
00109 #include <pthread.h>   // for pthread_mutex_t, pthread_cond_t
00110 #include <stdint.h>    // for uint32_t
00111 #include <sys/types.h> // for ssize_t
00112
00113 #ifdef __cplusplus
00114 extern "C"
00115 {
00116 #endif
00117
00139 #define PACKING_SZ 32
00140
00148   struct saurion
00149   {
00151     struct io_uring *rings;
00153     pthread_mutex_t *m_rings;
00155     int ss;
00157     int *efds;
00159     struct Node *list;
00161     pthread_mutex_t status_m;
00163     pthread_cond_t status_c;
00165     int status;
00167     struct threadpool *pool;
00169     uint32_t n_threads;
00171     uint32_t next;
00172
00181     struct saurion_callbacks
00182     {
00189       void (*on_connected) (const int fd, void *arg);
00191       void *on_connected_arg;
00192
00201       void (*on_readed) (const int fd, const void *const content,
00202                          const ssize_t len, void *arg);
00204       void *on_readed_arg;
00205
00212       void (*on_wrote) (const int fd, void *arg);
00213       void *on_wrote_arg;
00221       void (*on_closed) (const int fd, void *arg);
00223       void *on_closed_arg;
00224
00233       void (*on_error) (const int fd, const char *const content,
00234                         const ssize_t len, void *arg);
00236       void *on_error_arg;
00237     } __attribute__ ((aligned (PACKING_SZ))) cb;
00238   } __attribute__ ((aligned (PACKING_SZ)));
00239
00249   int saurion_set_socket (int p);
00250
00263   [[nodiscard]]
00264   struct saurion *saurion_create (uint32_t n_threads);
00265
00278   [[nodiscard]]
00279   int saurion_start (struct saurion *s);
00280
00291   void saurion_stop (const struct saurion *s);
00292
00305   void saurion_destroy (struct saurion *s);
00306
00319   void saurion_send (struct saurion *s, const int fd, const char *const msg);
00320
00321 #ifdef __cplusplus
00322 }
00323 #endif
00324
00325 #endif // !LOW_SAURION_H
00326
```

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

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

`#include <stdint.h>`
Include dependency graph for low_saurion_secret.h:

Functions

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

  Initializes a specified `iovec` structure with a message fragment.
- int set_request (struct request **r, struct Node **l, size_t s, const void *m, uint8_t h)

  Sets up a request and allocates iovec structures for data handling in liburing.
- int read_chunk (void **dest, size_t *len, struct request *const req)

  Reads a message chunk from the request's iovec buffers, handling messages that may span multiple iovec entries.
- void free_request (struct request *req, void **children_ptr, size_t amount)

7.6 low_saurion_secret.h

Go to the documentation of this file.

```
00001 #ifndef LOW_SAURION_SECRET_H
00002 #define LOW_SAURION_SECRET_H
00003
00004 #include <bits/types/struct_iovec.h>
00005 #include <stddef.h>
00006 #include <stdint.h>
00007
00008 #ifdef __cplusplus
00009 extern "C" {
00010 #endif
00015 #pragma GCC diagnostic push
00016 #pragma GCC diagnostic ignored "-Wpedantic"
00017 struct request {
00018   void *prev;
00019   size_t prev_size;
00020   size_t prev_remain;
00021   size_t next_iov;
00022   size_t next_offset;
00023   int event_type;
00024   size_t iovec_count;
00025   int client_socket;
00026   struct iovec iov[];
00027 };
00028 #pragma GCC diagnostic pop
00062 [[nodiscard]]
00063 int allocate_iovec(struct iovec *iov, size_t amount, size_t pos, size_t size, void **chd_ptr);
00064
00097 [[nodiscard]]
00098 int initialize_iovec(struct iovec *iov, size_t amount, size_t pos, const void *msg, size_t size,
00099                      uint8_t h);
00100
00127 [[nodiscard]]
00128 int set_request(struct request **r, struct Node **l, size_t s, const void *m, uint8_t h);
00129
00165 [[nodiscard]]
00166 int read_chunk(void **dest, size_t *len, struct request *const req);
00167
00168 void free_request(struct request *req, void **children_ptr, size_t amount);
00172 #ifdef __cplusplus
00173 }
00174 #endif
00175
00176 #endif  // !LOW_SAURION_SECRET_H
```

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

`#include <stdint.h>`
`#include <sys/types.h>`
Include dependency graph for saurion.hpp: This graph shows which files directly or indirectly include this file:

Classes

- class Saurion

7.8 saurion.hpp

Go to the documentation of this file.

```
00001 #ifndef SAURION_HPP
00002 #define SAURION_HPP
00003
00004 #include <stdint.h>    // for uint32_t
00005 #include <sys/types.h> // for ssize_t
00006
00007 class Saurion
00008 {
00009 public:
00010   using ConnectedCb = void (*) (const int, void *);
00011   using ReadedCb
00012       = void (*) (const int, const void *const, const ssize_t, void *);
00013   using WroteCb = void (*) (const int, void *);
00014   using ClosedCb = void (*) (const int, void *);
00015   using ErrorCb
00016       = void (*) (const int, const char *const, const ssize_t, void *);
00017
00018   explicit Saurion (const uint32_t thds, const int sck) noexcept;
00019   ~Saurion ();
00020
00021   Saurion (const Saurion &) = delete;
00022   Saurion (Saurion &&) = delete;
00023   Saurion &operator= (const Saurion &) = delete;
00024   Saurion &operator= (Saurion &&) = delete;
00025
00026   void init () noexcept;
00027   void stop () const noexcept;
00028
00029   Saurion *on_connected (ConnectedCb ncb, void *arg) noexcept;
00030   Saurion *on_readed (ReadedCb ncb, void *arg) noexcept;
00031   Saurion *on_wrote (WroteCb ncb, void *arg) noexcept;
00032   Saurion *on_closed (ClosedCb ncb, void *arg) noexcept;
00033   Saurion *on_error (ErrorCb ncb, void *arg) noexcept;
00034
00035   void send (const int fd, const char *const msg) noexcept;
00036
00037 private:
00038   struct saurion *s;
00039 };
00040
00041 #endif // !SAURION_HPP
```

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

`#include <stddef.h>`

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

Functions

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

7.10 threadpool.h

Go to the documentation of this file.

```
00001
00006 #ifndef THREADPOOL_H
00007 #define THREADPOOL_H
00008
00009 #include <stddef.h> // for size_t
00010
00011 #ifdef __cplusplus
00012 extern "C"
00013 {
00014 #endif
00015
00016   struct threadpool;
00017
00018   struct threadpool *threadpool_create (size_t num_threads);
00019
00020   struct threadpool *threadpool_create_default (void);
00021
00022   void threadpool_init (struct threadpool *pool);
00023
00024   void threadpool_add (struct threadpool *pool, void (*function) (void *),
00025                        void *argument);
00026
00027   void threadpool_stop (struct threadpool *pool);
00028
00029   int threadpool_empty (struct threadpool *pool);
00030
00031   void threadpool_wait_empty (struct threadpool *pool);
00032
00033   void threadpool_destroy (struct threadpool *pool);
00034
00035 #ifdef __cplusplus
00036 }
00037 #endif
00038
00039 #endif // !THREADPOOL_H
00040
```

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

```
#include "linked_list.h"
#include <pthread.h>
#include <stdlib.h>
```

Include dependency graph for linked_list.c:

Classes

- struct Node

Functions

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

Variables

- pthread_mutex_t list_mutex = PTHREAD_MUTEX_INITIALIZER

7.11.1 Function Documentation

7.11.1.1 create_node()

```
struct Node * create_node (
            void * ptr,
            size_t amount,
            void ** children )
```

Definition at line 17 of file linked_list.c.

```
00018 {
00019   struct Node *new_node = (struct Node *)malloc (sizeof (struct Node));
00020   if (!new_node)
00021     {
00022       return NULL;
00023     }
00024   new_node->ptr = ptr;
00025   new_node->size = amount;
00026   new_node->children = NULL;
00027   if (amount <= 0)
00028     {
00029       new_node->next = NULL;
00030       return new_node;
00031     }
00032   new_node->children
00033       = (struct Node **)malloc (sizeof (struct Node *) * amount);
00034   if (!new_node->children)
00035     {
00036       free (new_node);
00037       return NULL;
00038     }
00039   for (size_t i = 0; i < amount; ++i)
00040     {
00041       new_node->children[i] = (struct Node *)malloc (sizeof (struct Node));
00042
00043       if (!new_node->children[i])
00044         {
00045           for (size_t j = 0; j < i; ++j)
00046             {
00047               free (new_node->children[j]);
00048             }
00049           free (new_node);
00050           return NULL;
00051         }
00052     }
00053   for (size_t i = 0; i < amount; ++i)
00054     {
00055       new_node->children[i]->size = 0;
00056       new_node->children[i]->next = NULL;
00057       new_node->children[i]->ptr = children[i];
00058       new_node->children[i]->children = NULL;
00059     }
00060   new_node->next = NULL;
00061   return new_node;
00062 }
```

7.11.1.2 free_node()

```
void free_node (
            struct Node * current )
```

Definition at line 90 of file linked_list.c.

```
00091 {
00092   if (current->size > 0)
00093     {
00094       for (size_t i = 0; i < current->size; ++i)
00095         {
```

```
00096           free (current->children[i]->ptr);
00097           free (current->children[i]);
00098         }
00099       free (current->children);
00100     }
00101   free (current->ptr);
00102   free (current);
00103 }
```

7.11.1.3 list_delete_node()

```
void list_delete_node (
              struct Node ** head,
              const void *const ptr )
```

Definition at line 106 of file linked_list.c.

```
00107 {
00108   pthread_mutex_lock (&list_mutex);
00109   struct Node *current = *head;
00110   struct Node *prev = NULL;
00111
00112   if (current && current->ptr == ptr)
00113     {
00114       *head = current->next;
00115       free_node (current);
00116       pthread_mutex_unlock (&list_mutex);
00117       return;
00118     }
00119
00120   while (current && current->ptr != ptr)
00121     {
00122       prev = current;
00123       current = current->next;
00124     }
00125
00126   if (!current)
00127     {
00128       pthread_mutex_unlock (&list_mutex);
00129       return;
00130     }
00131
00132   prev->next = current->next;
00133   free_node (current);
00134   pthread_mutex_unlock (&list_mutex);
00135 }
```

7.11.1.4 list_free()

```
void list_free (
              struct Node ** head )
```

Definition at line 138 of file linked_list.c.

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

7.11.1.5 list_insert()

```
int list_insert (
              struct Node ** head,
              void * ptr,
              size_t amount,
              void ** children )
```

Definition at line 65 of file linked_list.c.

```
00066 {
00067   struct Node *new_node = create_node (ptr, amount, children);
00068   if (!new_node)
00069     {
00070       return 1;
00071     }
00072   pthread_mutex_lock (&list_mutex);
00073   if (!*head)
00074     {
00075       *head = new_node;
00076       pthread_mutex_unlock (&list_mutex);
00077       return 0;
00078     }
00079   struct Node *temp = *head;
00080   while (temp->next)
00081     {
00082       temp = temp->next;
00083     }
00084   temp->next = new_node;
00085   pthread_mutex_unlock (&list_mutex);
00086   return 0;
00087 }
```

7.11.2 Variable Documentation

7.11.2.1 list_mutex

```
pthread_mutex_t list_mutex = PTHREAD_MUTEX_INITIALIZER
```

Definition at line 14 of file linked_list.c.

7.12 linked_list.c

Go to the documentation of this file.

```
00001 #include "linked_list.h"
00002
00003 #include <pthread.h>
00004 #include <stdlib.h>
00005
00006 struct Node
00007 {
00008   void *ptr;
00009   size_t size;
00010   struct Node **children;
00011   struct Node *next;
00012 };
00013
00014 pthread_mutex_t list_mutex = PTHREAD_MUTEX_INITIALIZER;
00015
00016 struct Node *
00017 create_node (void *ptr, size_t amount, void **children)
00018 {
00019   struct Node *new_node = (struct Node *)malloc (sizeof (struct Node));
00020   if (!new_node)
00021     {
```

```
00022       return NULL;
00023     }
00024   new_node->ptr = ptr;
00025   new_node->size = amount;
00026   new_node->children = NULL;
00027   if (amount <= 0)
00028     {
00029       new_node->next = NULL;
00030       return new_node;
00031     }
00032   new_node->children
00033       = (struct Node **)malloc (sizeof (struct Node *) * amount);
00034   if (!new_node->children)
00035     {
00036       free (new_node);
00037       return NULL;
00038     }
00039   for (size_t i = 0; i < amount; ++i)
00040     {
00041       new_node->children[i] = (struct Node *)malloc (sizeof (struct Node));
00042
00043       if (!new_node->children[i])
00044         {
00045           for (size_t j = 0; j < i; ++j)
00046             {
00047               free (new_node->children[j]);
00048             }
00049           free (new_node);
00050           return NULL;
00051         }
00052     }
00053   for (size_t i = 0; i < amount; ++i)
00054     {
00055       new_node->children[i]->size = 0;
00056       new_node->children[i]->next = NULL;
00057       new_node->children[i]->ptr = children[i];
00058       new_node->children[i]->children = NULL;
00059     }
00060   new_node->next = NULL;
00061   return new_node;
00062 }
00063
00064 int
00065 list_insert (struct Node **head, void *ptr, size_t amount, void **children)
00066 {
00067   struct Node *new_node = create_node (ptr, amount, children);
00068   if (!new_node)
00069     {
00070       return 1;
00071     }
00072   pthread_mutex_lock (&list_mutex);
00073   if (!*head)
00074     {
00075       *head = new_node;
00076       pthread_mutex_unlock (&list_mutex);
00077       return 0;
00078     }
00079   struct Node *temp = *head;
00080   while (temp->next)
00081     {
00082       temp = temp->next;
00083     }
00084   temp->next = new_node;
00085   pthread_mutex_unlock (&list_mutex);
00086   return 0;
00087 }
00088
00089 void
00090 free_node (struct Node *current)
00091 {
00092   if (current->size > 0)
00093     {
00094       for (size_t i = 0; i < current->size; ++i)
00095         {
00096           free (current->children[i]->ptr);
00097           free (current->children[i]);
00098         }
00099       free (current->children);
00100     }
00101   free (current->ptr);
00102   free (current);
00103 }
00104
00105 void
00106 list_delete_node (struct Node **head, const void *const ptr)
00107 {
00108   pthread_mutex_lock (&list_mutex);
```

```
00109   struct Node *current = *head;
00110   struct Node *prev = NULL;
00111
00112   if (current && current->ptr == ptr)
00113     {
00114       *head = current->next;
00115       free_node (current);
00116       pthread_mutex_unlock (&list_mutex);
00117       return;
00118     }
00119
00120   while (current && current->ptr != ptr)
00121     {
00122       prev = current;
00123       current = current->next;
00124     }
00125
00126   if (!current)
00127     {
00128       pthread_mutex_unlock (&list_mutex);
00129       return;
00130     }
00131
00132   prev->next = current->next;
00133   free_node (current);
00134   pthread_mutex_unlock (&list_mutex);
00135 }
00136
00137 void
00138 list_free (struct Node **head)
00139 {
00140   pthread_mutex_lock (&list_mutex);
00141   struct Node *current = *head;
00142   struct Node *next;
00143
00144   while (current)
00145     {
00146       next = current->next;
00147       free_node (current);
00148       current = next;
00149     }
00150
00151   *head = NULL;
00152   pthread_mutex_unlock (&list_mutex);
00153 }
```

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

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

Include dependency graph for low_saurion.c:

Classes

- struct request
- struct saurion_wrapper
- struct chunk_params

Macros

- #define EV_ACC 0
- #define EV_REA 1
- #define EV_WRI 2
- #define EV_WAI 3
- #define EV_ERR 4
- #define MIN(a, b) ((a) < (b) ? (a) : (b))
- #define MAX(a, b) ((a) > (b) ? (a) : (b))

Functions

- static uint32_t next (struct saurion *s)
- static uint64_t htonll (uint64_t value)
- static uint64_t ntohll (uint64_t value)
- void free_request (struct request *req, void **children_ptr, size_t amount)
- int initialize_iovec (struct iovec *iov, size_t amount, size_t pos, const void *msg, size_t size, uint8_t h)

  Initializes a specified `iovec` structure with a message fragment.
- int allocate_iovec (struct iovec *iov, size_t amount, size_t pos, size_t size, void **chd_ptr)
- int set_request (struct request **r, struct Node **l, size_t s, const void *m, uint8_t h)

  Sets up a request and allocates iovec structures for data handling in liburing.
- static void add_accept (struct saurion *const s, struct sockaddr_in *const ca, socklen_t *const cal)
- static void add_fd (struct saurion *const s, int client_socket, int sel)
- static void add_efd (struct saurion *const s, const int client_socket, int sel)
- static void add_read (struct saurion *const s, const int client_socket)
- static void add_read_continue (struct saurion *const s, struct request *oreq, const int sel)
- static void add_write (struct saurion *const s, int fd, const char *const str, const int sel)
- static void handle_accept (const struct saurion *const s, const int fd)
- static size_t calculate_max_iov_content (const struct request *req)
- static int handle_previous_message (struct chunk_params *p)
- static int handle_partial_message (struct chunk_params *p)
- static int handle_new_message (struct chunk_params *p)
- static int prepare_destination (struct chunk_params *p)
- static void copy_data (struct chunk_params *p, uint8_t *ok)
- static uint8_t validate_and_update (struct chunk_params *p, uint8_t ok)
- static void read_chunk_free (struct chunk_params *p)
- int read_chunk (void **dest, size_t *len, struct request *const req)

  Reads a message chunk from the request's iovec buffers, handling messages that may span multiple iovec entries.
- static void handle_read (struct saurion *const s, struct request *const req)
- static void handle_write (const struct saurion *const s, const int fd)
- static void handle_error (const struct saurion *const s, const struct request *const req)
- static void handle_close (const struct saurion *const s, const struct request *const req)
- int saurion_set_socket (const int p)

  Creates a socket.
- struct saurion * saurion_create (uint32_t n_threads)

  Creates an instance of the `saurion` structure.

- static void handle_event_read (const struct io_uring_cqe *const cqe, struct saurion *const s, struct request *req)
- static int saurion_worker_master_loop_it (struct saurion *const s, struct sockaddr_in *client_addr, socklen_t *client_addr_len)
- void saurion_worker_master (void *arg)
- static int saurion_worker_slave_loop_it (struct saurion *const s, const int sel)
- void saurion_worker_slave (void *arg)
- int saurion_start (struct saurion *const s)

  Starts event processing in the `saurion` structure.
- void saurion_stop (const struct saurion *const s)

  Stops event processing in the `saurion` structure.
- void saurion_destroy (struct saurion *const s)

  Destroys the `saurion` structure and frees all associated resources.
- void saurion_send (struct saurion *const s, const int fd, const char *const msg)

  Sends a message through a socket using io_uring.

Variables

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

7.13.1 Macro Definition Documentation

7.13.1.1 EV_ACC

```
#define EV_ACC 0
```

Definition at line 26 of file low_saurion.c.

7.13.1.2 EV_ERR

```
#define EV_ERR 4
```

Definition at line 30 of file low_saurion.c.

7.13.1.3 EV_REA

```
#define EV_REA 1
```

Definition at line 27 of file low_saurion.c.

7.13.1.4 EV_WAI

```
#define EV_WAI 3
```

Definition at line 29 of file low_saurion.c.

7.13.1.5 EV_WRI

```
#define EV_WRI 2
```

Definition at line 28 of file low_saurion.c.

7.13.1.6 MAX

```
#define MAX(
              a,
              b ) ((a) > (b) ?  (a) :  (b))
```

Definition at line 46 of file low_saurion.c.

7.13.1.7 MIN

```
#define MIN(
              a,
              b ) ((a) < (b) ?  (a) :  (b))
```

Definition at line 45 of file low_saurion.c.

7.13.2 Function Documentation

7.13.2.1 add_accept()

```
static void add_accept (
            struct saurion *const s,
            struct sockaddr_in *const ca,
            socklen_t *const cal )  [inline], [static]
```

Definition at line 257 of file low_saurion.c.

```
00259 {
00260   int res = ERROR_CODE;
00261   pthread_mutex_lock (&s->m_rings[0]);
00262   while (res != SUCCESS_CODE)
00263     {
00264       struct io_uring_sqe *sqe = io_uring_get_sqe (&s->rings[0]);
00265       while (!sqe)
00266         {
00267           sqe = io_uring_get_sqe (&s->rings[0]);
00268           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00269         }
00270       struct request *req = NULL;
00271       if (!set_request (&req, &s->list, 0, NULL, 0))
00272         {
00273           free (sqe);
00274           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00275           res = ERROR_CODE;
00276           continue;
00277         }
00278       req->client_socket = 0;
00279       req->event_type = EV_ACC;
00280       io_uring_prep_accept (sqe, s->ss, (struct sockaddr *const)ca, cal, 0);
00281       io_uring_sqe_set_data (sqe, req);
00282       if (io_uring_submit (&s->rings[0]) < 0)
00283         {
00284           free (sqe);
00285           list_delete_node (&s->list, req);
00286           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00287           res = ERROR_CODE;
00288           continue;
00289         }
00290       res = SUCCESS_CODE;
00291     }
00292   pthread_mutex_unlock (&s->m_rings[0]);
00293 }
```

7.13.2.2 add_efd()

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

Definition at line 336 of file low_saurion.c.

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

7.13.2.3 add_fd()

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

Definition at line 297 of file low_saurion.c.

```
00298 {
00299   int res = ERROR_CODE;
00300   pthread_mutex_lock (&s->m_rings[sel]);
00301   while (res != SUCCESS_CODE)
00302     {
00303       struct io_uring *ring = &s->rings[sel];
00304       struct io_uring_sqe *sqe = io_uring_get_sqe (ring);
00305       while (!sqe)
00306         {
00307           sqe = io_uring_get_sqe (ring);
00308           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00309         }
00310       struct request *req = NULL;
00311       if (!set_request (&req, &s->list, CHUNK_SZ, NULL, 0))
00312         {
00313           free (sqe);
00314           res = ERROR_CODE;
00315           continue;
00316         }
00317       req->event_type = EV_REA;
00318       req->client_socket = client_socket;
00319       io_uring_prep_readv (sqe, client_socket, &req->iov[0], req->iovec_count,
00320                            0);
00321       io_uring_sqe_set_data (sqe, req);
00322       if (io_uring_submit (ring) < 0)
00323         {
00324           free (sqe);
00325           list_delete_node (&s->list, req);
00326           res = ERROR_CODE;
00327           continue;
00328         }
00329       res = SUCCESS_CODE;
00330     }
00331   pthread_mutex_unlock (&s->m_rings[sel]);
00332 }
```

7.13.2.4 add_read()

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

Definition at line 343 of file low_saurion.c.

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

7.13.2.5 add_read_continue()

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

Definition at line 351 of file low_saurion.c.

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

```
00364         }
00365       if (!set_request (&oreq, &s->list, oreq->prev_remain, NULL, 0))
00366         {
00367           free (sqe);
00368           res = ERROR_CODE;
00369           continue;
00370         }
00371       io_uring_prep_readv (sqe, oreq->client_socket, &oreq->iov[0],
00372                            oreq->iovec_count, 0);
00373       io_uring_sqe_set_data (sqe, oreq);
00374       if (io_uring_submit (ring) < 0)
00375         {
00376           free (sqe);
00377           list_delete_node (&s->list, oreq);
00378           res = ERROR_CODE;
00379           continue;
00380         }
00381       res = SUCCESS_CODE;
00382     }
00383   pthread_mutex_unlock (&s->m_rings[sel]);
00384 }
```

7.13.2.6 add_write()

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

Definition at line 388 of file low_saurion.c.

```
00390 {
00391   int res = ERROR_CODE;
00392   pthread_mutex_lock (&s->m_rings[sel]);
00393   while (res != SUCCESS_CODE)
00394     {
00395       struct io_uring *ring = &s->rings[sel];
00396       struct io_uring_sqe *sqe = io_uring_get_sqe (ring);
00397       while (!sqe)
00398         {
00399           sqe = io_uring_get_sqe (ring);
00400           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00401         }
00402       struct request *req = NULL;
00403       if (!set_request (&req, &s->list, strlen (str), (const void *const)str,
00404                         1))
00405         {
00406           free (sqe);
00407           res = ERROR_CODE;
00408           continue;
00409         }
00410       req->event_type = EV_WRI;
00411       req->client_socket = fd;
00412       io_uring_prep_writev (sqe, req->client_socket, req->iov,
00413                             req->iovec_count, 0);
00414       io_uring_sqe_set_data (sqe, req);
00415       if (io_uring_submit (ring) < 0)
00416         {
00417           free (sqe);
00418           list_delete_node (&s->list, req);
00419           res = ERROR_CODE;
00420           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00421           continue;
00422         }
00423       res = SUCCESS_CODE;
00424     }
00425   pthread_mutex_unlock (&s->m_rings[sel]);
00426 }
```

7.13.2.7 calculate_max_iov_content()

```
static size_t calculate_max_iov_content (
            const struct request * req )  [inline], [static]
```

Definition at line 442 of file low_saurion.c.

```
00443 {
00444   size_t max_iov_cont = 0;
00445   for (size_t i = 0; i < req->iovec_count; ++i)
00446     {
00447       max_iov_cont += req->iov[i].iov_len;
00448     }
00449   return max_iov_cont;
00450 }
```

7.13.2.8 copy_data()

```
static void copy_data (
            struct chunk_params * p,
            uint8_t * ok )  [inline], [static]
```

Definition at line 584 of file low_saurion.c.

```
00585 {
00586   size_t curr_iov_msg_rem = 0;
00587   *ok = 1UL;
00588   while (1)
00589     {
00590       curr_iov_msg_rem = MIN (
00591           p->cont_rem, (p->req->iov[p->curr_iov].iov_len - p->curr_iov_off));
00592       memcpy ((uint8_t *)p->dest_ptr + p->dest_off,
00593               (uint8_t *)p->req->iov[p->curr_iov].iov_base + p->curr_iov_off,
00594               curr_iov_msg_rem);
00595       p->dest_off += curr_iov_msg_rem;
00596       p->curr_iov_off += curr_iov_msg_rem;
00597       p->cont_rem -= curr_iov_msg_rem;
00598
00599       if (p->cont_rem <= 0)
00600         {
00601           if (*(((uint8_t *)p->req->iov[p->curr_iov].iov_base)
00602                 + p->curr_iov_off)
00603               != 0)
00604             {
00605               *ok = 0UL;
00606             }
00607           *p->len = p->cont_sz;
00608           ++p->curr_iov_off;
00609           break;
00610         }
00611       if (p->curr_iov_off >= (p->req->iov[p->curr_iov].iov_len))
00612         {
00613           ++p->curr_iov;
00614           if (p->curr_iov == p->req->iovec_count)
00615             {
00616               break;
00617             }
00618           p->curr_iov_off = 0;
00619         }
00620     }
00621 }
```

7.13.2.9 handle_accept()

```
static void handle_accept (
            const struct saurion *const s,
            const int fd )  [inline], [static]
```

Definition at line 431 of file low_saurion.c.

```
00432 {
00433   if (s->cb.on_connected)
00434     {
00435       s->cb.on_connected (fd, s->cb.on_connected_arg);
00436     }
00437 }
```

7.13.2.10 handle_close()

```
static void handle_close (
            const struct saurion *const s,
            const struct request *const req )  [inline], [static]
```

Definition at line 785 of file low_saurion.c.

```
00786 {
00787   if (s->cb.on_closed)
00788     {
00789       s->cb.on_closed (req->client_socket, s->cb.on_closed_arg);
00790     }
00791   close (req->client_socket);
00792 }
```

7.13.2.11 handle_error()

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

Definition at line 773 of file low_saurion.c.

```
00774 {
00775   if (s->cb.on_error)
00776     {
00777       const char *resp = "ERROR";
00778       s->cb.on_error (req->client_socket, resp, (ssize_t)strlen (resp),
00779                       s->cb.on_error_arg);
00780     }
00781 }
```

7.13.2.12 handle_event_read()

```
static void handle_event_read (
             const struct io_uring_cqe *const cqe,
             struct saurion *const s,
             struct request * req )  [inline], [static]
```

Definition at line 951 of file low_saurion.c.

```
00953 {
00954   if (cqe->res < 0)
00955     {
00956       handle_error (s, req);
00957     }
00958   if (cqe->res < 1)
00959     {
00960       handle_close (s, req);
00961     }
00962   if (cqe->res > 0)
00963     {
00964       handle_read (s, req);
00965     }
00966   list_delete_node (&s->list, req);
00967 }
```

7.13.2.13 handle_new_message()

```
static int handle_new_message (
             struct chunk_params * p )  [inline], [static]
```

Definition at line 532 of file low_saurion.c.

```
00533 {
00534   p->curr_iov = 0;
00535   p->curr_iov_off = 0;
00536
00537   p->cont_sz = *(size_t *)((uint8_t *)p->req->iov[p->curr_iov].iov_base
00538                            + p->curr_iov_off);
00539   p->cont_sz = ntohll (p->cont_sz);
00540   p->curr_iov_off += sizeof (uint64_t);
00541   p->cont_rem = p->cont_sz;
00542   p->dest_off = p->cont_sz - p->cont_rem;
00543
00544   if (p->cont_rem <= p->max_iov_cont)
00545     {
00546       *p->dest = malloc (p->cont_sz);
00547       if (!*p->dest)
00548         {
00549           return ERROR_CODE; // Error al asignar memoria.
00550         }
00551       p->dest_ptr = *p->dest;
00552     }
00553   else
00554     {
00555       p->req->prev = malloc (p->cont_sz);
00556       if (!p->req->prev)
00557         {
00558           return ERROR_CODE; // Error al asignar memoria.
00559         }
00560       p->dest_ptr = p->req->prev;
00561       *p->dest = NULL;
00562     }
00563   return SUCCESS_CODE;
00564 }
```

7.13.2.14 handle_partial_message()

```
static int handle_partial_message (
            struct chunk_params * p )  [inline], [static]
```

Definition at line 494 of file low_saurion.c.

```
00495 {
00496   p->curr_iov = p->req->next_iov;
00497   p->curr_iov_off = p->req->next_offset;
00498
00499   p->cont_sz = *(size_t *)((uint8_t *)p->req->iov[p->curr_iov].iov_base
00500                            + p->curr_iov_off);
00501   p->cont_sz = ntohll (p->cont_sz);
00502   p->curr_iov_off += sizeof (uint64_t);
00503   p->cont_rem = p->cont_sz;
00504   p->dest_off = p->cont_sz - p->cont_rem;
00505
00506   if ((p->curr_iov_off + p->cont_rem + 1) <= p->max_iov_cont)
00507     {
00508       *p->dest = malloc (p->cont_sz);
00509       if (!*p->dest)
00510         {
00511           return ERROR_CODE;
00512         }
00513       p->dest_ptr = *p->dest;
00514     }
00515   else
00516     {
00517       p->req->prev = malloc (p->cont_sz);
00518       if (!p->req->prev)
00519         {
00520           return ERROR_CODE;
00521         }
00522       p->dest_ptr = p->req->prev;
00523       *p->dest = NULL;
00524       *p->len = 0;
00525     }
00526   return SUCCESS_CODE;
00527 }
```

7.13.2.15 handle_previous_message()

```
static int handle_previous_message (
            struct chunk_params * p )  [inline], [static]
```

Definition at line 469 of file low_saurion.c.

```
00470 {
00471   p->cont_sz = p->req->prev_size;
00472   p->cont_rem = p->req->prev_remain;
00473   p->dest_off = p->cont_sz - p->cont_rem;
00474
00475   if (p->cont_rem <= p->max_iov_cont)
00476     {
00477       *p->dest = p->req->prev;
00478       p->dest_ptr = *p->dest;
00479       p->req->prev = NULL;
00480       p->req->prev_size = 0;
00481       p->req->prev_remain = 0;
00482     }
00483   else
00484     {
00485       p->dest_ptr = p->req->prev;
00486       *p->dest = NULL;
00487     }
00488   return SUCCESS_CODE;
00489 }
```

7.13.2.16 handle_read()

```
static void handle_read (
            struct saurion *const s,
            struct request *const req )  [inline], [static]
```

Definition at line 724 of file low_saurion.c.

```
00725 {
00726   void *msg = NULL;
00727   size_t len = 0;
00728   while (1)
00729     {
00730       if (!read_chunk (&msg, &len, req))
00731         {
00732           break;
00733         }
00734       if (req->next_iov || req->next_offset)
00735         {
00736           if (s->cb.on_readed && msg)
00737             {
00738               s->cb.on_readed (req->client_socket, msg, len,
00739                                s->cb.on_readed_arg);
00740             }
00741           free (msg);
00742           msg = NULL;
00743           continue;
00744         }
00745       if (req->prev && req->prev_size && req->prev_remain)
00746         {
00747           add_read_continue (s, req, next (s));
00748           return;
00749         }
00750       if (s->cb.on_readed && msg)
00751         {
00752           s->cb.on_readed (req->client_socket, msg, len, s->cb.on_readed_arg);
00753         }
00754       free (msg);
00755       msg = NULL;
00756       break;
00757     }
00758   add_read (s, req->client_socket);
00759 }
```

7.13.2.17 handle_write()

```
static void handle_write (
            const struct saurion *const s,
            const int fd )  [inline], [static]
```

Definition at line 763 of file low_saurion.c.

```
00764 {
00765   if (s->cb.on_wrote)
00766     {
00767       s->cb.on_wrote (fd, s->cb.on_wrote_arg);
00768     }
00769 }
```

7.13.2.18 htonll()

```
static uint64_t htonll (
            uint64_t value )  [inline], [static]
```

Definition at line 66 of file low_saurion.c.

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

```
00069   if (*(char *)&num == 42)
00070     {
00071       uint32_t high_part = htonl ((uint32_t)(value » 32));
00072       uint32_t low_part = htonl ((uint32_t)(value & 0xFFFFFFFFLL));
00073       return ((uint64_t)low_part « 32) | high_part;
00074     }
00075   return value;
00076 }
```

7.13.2.19 next()

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

Definition at line 58 of file low_saurion.c.

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

7.13.2.20 ntohll()

```
static uint64_t ntohll (
             uint64_t value )  [inline], [static]
```

Definition at line 80 of file low_saurion.c.

```
00081 {
00082   int num = 42;
00083   if (*(char *)&num == 42)
00084     {
00085       uint32_t high_part = ntohl ((uint32_t)(value » 32));
00086       uint32_t low_part = ntohl ((uint32_t)(value & 0xFFFFFFFFLL));
00087       return ((uint64_t)low_part « 32) | high_part;
00088     }
00089   return value;
00090 }
```

7.13.2.21 prepare_destination()

```
static int prepare_destination (
             struct chunk_params * p )  [inline], [static]
```

Definition at line 569 of file low_saurion.c.

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

7.13.2.22 read_chunk_free()

```
static void read_chunk_free (
            struct chunk_params * p )  [inline], [static]
```

Definition at line 662 of file low_saurion.c.

```
00663 {
00664   free (p->dest_ptr);
00665   p->dest_ptr = NULL;
00666   *p->dest = NULL;
00667   *p->len = 0;
00668   p->req->next_iov = 0;
00669   p->req->next_offset = 0;
00670   for (size_t i = p->curr_iov; i < p->req->iovec_count; ++i)
00671     {
00672       for (size_t j = p->curr_iov_off; j < p->req->iov[i].iov_len; ++j)
00673         {
00674           uint8_t foot = *((uint8_t *)p->req->iov[i].iov_base) + j;
00675           if (foot == 0)
00676             {
00677               p->req->next_iov = i;
00678               p->req->next_offset = (j + 1) % p->req->iov[i].iov_len;
00679               return;
00680             }
00681         }
00682     }
00683 }
```

7.13.2.23 saurion_worker_master()

```
void saurion_worker_master (
            void * arg )
```

Definition at line 1029 of file low_saurion.c.

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

7.13.2.24 saurion_worker_master_loop_it()

```
static int saurion_worker_master_loop_it (
              struct saurion *const s,
              struct sockaddr_in * client_addr,
              socklen_t * client_addr_len )  [inline], [static]
```

Definition at line 972 of file low_saurion.c.

```
00975 {
00976   LOG_INIT (" ");
00977   struct io_uring ring = s->rings[0];
00978   struct io_uring_cqe *cqe = NULL;
00979   int ret = io_uring_wait_cqe (&ring, &cqe);
00980   if (ret < 0)
00981     {
00982       free (cqe);
00983       LOG_END (" ");
00984       return CRITICAL_CODE;
00985     }
00986   struct request *req = (struct request *)cqe->user_data;
00987   if (!req)
00988     {
00989       io_uring_cqe_seen (&s->rings[0], cqe);
00990       LOG_END (" ");
00991       return SUCCESS_CODE;
00992     }
00993   if (cqe->res < 0)
00994     {
00995       list_delete_node (&s->list, req);
00996       LOG_END (" ");
00997       return CRITICAL_CODE;
00998     }
00999   if (req->client_socket == s->efds[0])
01000     {
01001       io_uring_cqe_seen (&s->rings[0], cqe);
01002       list_delete_node (&s->list, req);
01003       LOG_END (" ");
01004       return ERROR_CODE;
01005     }
01006   io_uring_cqe_seen (&s->rings[0], cqe);
01007   switch (req->event_type)
01008     {
01009     case EV_ACC:
01010       handle_accept (s, cqe->res);
01011       add_accept (s, client_addr, client_addr_len);
01012       add_read (s, cqe->res);
01013       list_delete_node (&s->list, req);
01014       break;
01015     case EV_REA:
01016       handle_event_read (cqe, s, req);
01017       break;
01018     case EV_WRI:
01019       handle_write (s, req->client_socket);
01020       list_delete_node (&s->list, req);
01021       break;
01022     }
01023   LOG_END (" ");
01024   return SUCCESS_CODE;
01025 }
```

7.13.2.25 saurion_worker_slave()

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

Definition at line 1114 of file low_saurion.c.

```
01115 {
01116   LOG_INIT (" ");
01117   struct saurion_wrapper *const ss = (struct saurion_wrapper *)arg;
01118   struct saurion *s = ss->s;
01119   const int sel = ss->sel;
01120   free (ss);
01121
01122   add_efd (s, s->efds[sel], sel);
```

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

7.13.2.26 saurion_worker_slave_loop_it()

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

Definition at line 1063 of file low_saurion.c.

```
01064 {
01065   LOG_INIT (" ");
01066   struct io_uring ring = s->rings[sel];
01067   struct io_uring_cqe *cqe = NULL;
01068
01069   add_efd (s, s->efds[sel], sel);
01070   int ret = io_uring_wait_cqe (&ring, &cqe);
01071   if (ret < 0)
01072     {
01073       free (cqe);
01074       LOG_END (" ");
01075       return CRITICAL_CODE;
01076     }
01077   struct request *req = (struct request *)cqe->user_data;
01078   if (!req)
01079     {
01080       io_uring_cqe_seen (&ring, cqe);
01081       LOG_END (" ");
01082       return SUCCESS_CODE;
01083     }
01084   if (cqe->res < 0)
01085     {
01086       list_delete_node (&s->list, req);
01087       LOG_END (" ");
01088       return CRITICAL_CODE;
01089     }
01090   if (req->client_socket == s->efds[sel])
01091     {
01092       io_uring_cqe_seen (&ring, cqe);
01093       list_delete_node (&s->list, req);
01094       LOG_END (" ");
01095       return ERROR_CODE;
01096     }
01097   io_uring_cqe_seen (&ring, cqe);
01098   switch (req->event_type)
01099     {
01100     case EV_REA:
01101       handle_event_read (cqe, s, req);
01102       break;
01103     case EV_WRI:
01104       handle_write (s, req->client_socket);
01105       list_delete_node (&s->list, req);
01106       break;
01107     }
01108   LOG_END (" ");
01109   return SUCCESS_CODE;
01110 }
```

7.13.2.27 validate_and_update()

```
static uint8_t validate_and_update (
            struct chunk_params * p,
            uint8_t ok )  [inline], [static]
```

Definition at line 626 of file low_saurion.c.

```
00627 {
00628   if (p->req->prev)
00629     {
00630       p->req->prev_size = p->cont_sz;
00631       p->req->prev_remain = p->cont_rem;
00632       *p->dest = NULL;
00633       *p->len = 0;
00634     }
00635   else
00636     {
00637       p->req->prev_size = 0;
00638       p->req->prev_remain = 0;
00639     }
00640   if (p->curr_iov < p->req->iovec_count)
00641     {
00642       uint64_t next_sz
00643           = *(uint64_t *)(((uint8_t *)p->req->iov[p->curr_iov].iov_base)
00644                           + p->curr_iov_off);
00645       if ((p->req->iov[p->curr_iov].iov_len > p->curr_iov_off) && next_sz)
00646         {
00647           p->req->next_iov = p->curr_iov;
00648           p->req->next_offset = p->curr_iov_off;
00649         }
00650       else
00651         {
00652           p->req->next_iov = 0;
00653           p->req->next_offset = 0;
00654         }
00655     }
00656
00657   return ok ?  SUCCESS_CODE : ERROR_CODE;
00658 }
```

7.13.3 Variable Documentation

7.13.3.1 TIMEOUT_RETRY_SPEC

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

Definition at line 48 of file low_saurion.c.

7.14 low_saurion.c

Go to the documentation of this file.

```
00001 #include "low_saurion.h"
00002 #include "config.h"      // for ERROR_CODE, SUCCESS_CODE, CHUNK_SZ
00003 #include "linked_list.h" // for list_delete_node, list_free, list_insert
00004 #include "threadpool.h"  // for threadpool_add, threadpool_create
00005
00006 #include <arpa/inet.h>             // for htonl, ntohl, htons
00007 #include <bits/socket-constants.h> // for SOL_SOCKET, SO_REUSEADDR
00008 #include <liburing.h>          // for io_uring_get_sqe, io_uring, io_uring_...
00009 #include <liburing/io_uring.h> // for io_uring_cqe
00010 #include <nanologger.h>        // for LOG_END, LOG_INIT
00011 #include <netinet/in.h>        // for sockaddr_in, INADDR_ANY, in_addr
00012 #include <pthread.h>           // for pthread_mutex_lock, pthread_mutex_unlock
00013 #include <stdint.h>            // for uint32_t, uint64_t, uint8_t
00014 #include <stdio.h>             // for NULL
```

```
00015 #include <stdlib.h>            // for free, malloc
00016 #include <string.h>            // for memset, memcpy, strlen
00017 #include <sys/eventfd.h>       // for eventfd, EFD_NONBLOCK
00018 #include <sys/socket.h>        // for socklen_t, bind, listen, setsockopt
00019 #include <sys/uio.h>           // for iovec
00020 #include <time.h>              // for nanosleep
00021 #include <unistd.h>            // for close, write
00022
00023 struct Node;
00024 struct iovec;
00025
00026 #define EV_ACC 0
00027 #define EV_REA 1
00028 #define EV_WRI 2
00029 #define EV_WAI 3
00030 #define EV_ERR 4
00031
00032 struct request
00033 {
00034   void *prev;
00035   size_t prev_size;
00036   size_t prev_remain;
00037   size_t next_iov;
00038   size_t next_offset;
00039   int event_type;
00040   size_t iovec_count;
00041   int client_socket;
00042   struct iovec iov[];
00043 };
00044
00045 #define MIN(a, b) ((a) < (b) ?  (a) :  (b))
00046 #define MAX(a, b) ((a) > (b) ?  (a) :  (b))
00047
00048 static struct timespec TIMEOUT_RETRY_SPEC = { 0, TIMEOUT_RETRY * 1000L };
00049
00050 struct saurion_wrapper
00051 {
00052   struct saurion *s;
00053   uint32_t sel;
00054 };
00055
00056 // next
00057 static inline uint32_t
00058 next (struct saurion *s)
00059 {
00060   s->next = (s->next + 1) % s->n_threads;
00061   return s->next;
00062 }
00063
00064 // htonll
00065 static inline uint64_t
00066 htonll (uint64_t value)
00067 {
00068   int num = 42;
00069   if (*(char *)&num == 42)
00070     {
00071       uint32_t high_part = htonl ((uint32_t)(value » 32));
00072       uint32_t low_part = htonl ((uint32_t)(value & 0xFFFFFFFFLL));
00073       return ((uint64_t)low_part « 32) | high_part;
00074     }
00075   return value;
00076 }
00077
00078 // ntohll
00079 static inline uint64_t
00080 ntohll (uint64_t value)
00081 {
00082   int num = 42;
00083   if (*(char *)&num == 42)
00084     {
00085       uint32_t high_part = ntohl ((uint32_t)(value » 32));
00086       uint32_t low_part = ntohl ((uint32_t)(value & 0xFFFFFFFFLL));
00087       return ((uint64_t)low_part « 32) | high_part;
00088     }
00089   return value;
00090 }
00091
00092 // free_request
00093 void
00094 free_request (struct request *req, void **children_ptr, size_t amount)
00095 {
00096   if (children_ptr)
00097     {
00098       free (children_ptr);
00099       children_ptr = NULL;
00100     }
00101   for (size_t i = 0; i < amount; ++i)
```

```
00102     {
00103       free (req->iov[i].iov_base);
00104       req->iov[i].iov_base = NULL;
00105     }
00106   free (req);
00107   req = NULL;
00108   free (children_ptr);
00109   children_ptr = NULL;
00110 }
00111
00112 // initialize_iovec
00113 [[nodiscard]]
00114 int
00115 initialize_iovec (struct iovec *iov, size_t amount, size_t pos,
00116                   const void *msg, size_t size, uint8_t h)
00117 {
00118   if (!iov || !iov->iov_base)
00119     {
00120       return ERROR_CODE;
00121     }
00122   if (msg)
00123     {
00124       size_t len = iov->iov_len;
00125       char *dest = (char *)iov->iov_base;
00126       char *orig = (char *)msg + pos * CHUNK_SZ;
00127       size_t cpy_sz = 0;
00128       if (h)
00129         {
00130           if (pos == 0)
00131             {
00132               uint64_t send_size = htonll (size);
00133               memcpy (dest, &send_size, sizeof (uint64_t));
00134               dest += sizeof (uint64_t);
00135               len -= sizeof (uint64_t);
00136             }
00137           else
00138             {
00139               orig -= sizeof (uint64_t);
00140             }
00141           if ((pos + 1) == amount)
00142             {
00143               --len;
00144               cpy_sz = (len < size ?  len :  size);
00145               dest[cpy_sz] = 0;
00146             }
00147         }
00148       cpy_sz = (len < size ?  len :  size);
00149       memcpy (dest, orig, cpy_sz);
00150       dest += cpy_sz;
00151       size_t rem = CHUNK_SZ - (dest - (char *)iov->iov_base);
00152       memset (dest, 0, rem);
00153     }
00154   else
00155     {
00156       memset ((char *)iov->iov_base, 0, CHUNK_SZ);
00157     }
00158   return SUCCESS_CODE;
00159 }
00160
00161 // allocate_iovec
00162 [[nodiscard]]
00163 int
00164 allocate_iovec (struct iovec *iov, size_t amount, size_t pos, size_t size,
00165                 void **chd_ptr)
00166 {
00167   if (!iov || !chd_ptr)
00168     {
00169       return ERROR_CODE;
00170     }
00171   iov->iov_base = malloc (CHUNK_SZ);
00172   if (!iov->iov_base)
00173     {
00174       return ERROR_CODE;
00175     }
00176   iov->iov_len = (pos == (amount - 1) ?  (size % CHUNK_SZ) : CHUNK_SZ);
00177   if (iov->iov_len == 0)
00178     {
00179       iov->iov_len = CHUNK_SZ;
00180     }
00181   chd_ptr[pos] = iov->iov_base;
00182   return SUCCESS_CODE;
00183 }
00184
00185 // set_request
00186 [[nodiscard]]
00187 int
00188 set_request (struct request **r, struct Node **l, size_t s, const void *m,
```

```
00189              uint8_t h)
00190 {
00191   uint64_t full_size = s;
00192   if (h)
00193     {
00194       full_size += (sizeof (uint64_t) + sizeof (uint8_t));
00195     }
00196   size_t amount = full_size / CHUNK_SZ;
00197   amount = amount + (full_size % CHUNK_SZ == 0 ?  0 :  1);
00198   struct request *temp = (struct request *)malloc (
00199       sizeof (struct request) + sizeof (struct iovec) * amount);
00200   if (!temp)
00201     {
00202       return ERROR_CODE;
00203     }
00204   if (!*r)
00205     {
00206       *r = temp;
00207       (*r)->prev = NULL;
00208       (*r)->prev_size = 0;
00209       (*r)->prev_remain = 0;
00210       (*r)->next_iov = 0;
00211       (*r)->next_offset = 0;
00212     }
00213   else
00214     {
00215       temp->client_socket = (*r)->client_socket;
00216       temp->event_type = (*r)->event_type;
00217       temp->prev = (*r)->prev;
00218       temp->prev_size = (*r)->prev_size;
00219       temp->prev_remain = (*r)->prev_remain;
00220       temp->next_iov = (*r)->next_iov;
00221       temp->next_offset = (*r)->next_offset;
00222       *r = temp;
00223     }
00224   struct request *req = *r;
00225   req->iovec_count = (int)amount;
00226   void **children_ptr = (void **)malloc (amount * sizeof (void *));
00227   if (!children_ptr)
00228     {
00229       free_request (req, children_ptr, 0);
00230       return ERROR_CODE;
00231     }
00232   for (size_t i = 0; i < amount; ++i)
00233     {
00234       if (!allocate_iovec (&req->iov[i], amount, i, full_size, children_ptr))
00235         {
00236           free_request (req, children_ptr, amount);
00237           return ERROR_CODE;
00238         }
00239       if (!initialize_iovec (&req->iov[i], amount, i, m, s, h))
00240         {
00241           free_request (req, children_ptr, amount);
00242           return ERROR_CODE;
00243         }
00244     }
00245   if (list_insert (l, req, amount, children_ptr))
00246     {
00247       free_request (req, children_ptr, amount);
00248       return ERROR_CODE;
00249     }
00250   free (children_ptr);
00251   return SUCCESS_CODE;
00252 }
00253
00254 /****************** ADDERS ******************/
00255 // add_accept
00256 static inline void
00257 add_accept (struct saurion *const s, struct sockaddr_in *const ca,
00258             socklen_t *const cal)
00259 {
00260   int res = ERROR_CODE;
00261   pthread_mutex_lock (&s->m_rings[0]);
00262   while (res != SUCCESS_CODE)
00263     {
00264       struct io_uring_sqe *sqe = io_uring_get_sqe (&s->rings[0]);
00265       while (!sqe)
00266         {
00267           sqe = io_uring_get_sqe (&s->rings[0]);
00268           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00269         }
00270       struct request *req = NULL;
00271       if (!set_request (&req, &s->list, 0, NULL, 0))
00272         {
00273           free (sqe);
00274           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00275           res = ERROR_CODE;
```

```
00276           continue;
00277         }
00278       req->client_socket = 0;
00279       req->event_type = EV_ACC;
00280       io_uring_prep_accept (sqe, s->ss, (struct sockaddr *const)ca, cal, 0);
00281       io_uring_sqe_set_data (sqe, req);
00282       if (io_uring_submit (&s->rings[0]) < 0)
00283         {
00284           free (sqe);
00285           list_delete_node (&s->list, req);
00286           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00287           res = ERROR_CODE;
00288           continue;
00289         }
00290       res = SUCCESS_CODE;
00291     }
00292   pthread_mutex_unlock (&s->m_rings[0]);
00293 }
00294
00295 // add_fd
00296 static inline void
00297 add_fd (struct saurion *const s, int client_socket, int sel)
00298 {
00299   int res = ERROR_CODE;
00300   pthread_mutex_lock (&s->m_rings[sel]);
00301   while (res != SUCCESS_CODE)
00302     {
00303       struct io_uring *ring = &s->rings[sel];
00304       struct io_uring_sqe *sqe = io_uring_get_sqe (ring);
00305       while (!sqe)
00306         {
00307           sqe = io_uring_get_sqe (ring);
00308           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00309         }
00310       struct request *req = NULL;
00311       if (!set_request (&req, &s->list, CHUNK_SZ, NULL, 0))
00312         {
00313           free (sqe);
00314           res = ERROR_CODE;
00315           continue;
00316         }
00317       req->event_type = EV_REA;
00318       req->client_socket = client_socket;
00319       io_uring_prep_readv (sqe, client_socket, &req->iov[0], req->iovec_count,
00320                            0);
00321       io_uring_sqe_set_data (sqe, req);
00322       if (io_uring_submit (ring) < 0)
00323         {
00324           free (sqe);
00325           list_delete_node (&s->list, req);
00326           res = ERROR_CODE;
00327           continue;
00328         }
00329       res = SUCCESS_CODE;
00330     }
00331   pthread_mutex_unlock (&s->m_rings[sel]);
00332 }
00333
00334 // add_efd
00335 static inline void
00336 add_efd (struct saurion *const s, const int client_socket, int sel)
00337 {
00338   add_fd (s, client_socket, sel);
00339 }
00340
00341 // add_read
00342 static inline void
00343 add_read (struct saurion *const s, const int client_socket)
00344 {
00345   int sel = next (s);
00346   add_fd (s, client_socket, sel);
00347 }
00348
00349 // add_read_continue
00350 static inline void
00351 add_read_continue (struct saurion *const s, struct request *oreq,
00352                    const int sel)
00353 {
00354   pthread_mutex_lock (&s->m_rings[sel]);
00355   int res = ERROR_CODE;
00356   while (res != SUCCESS_CODE)
00357     {
00358       struct io_uring *ring = &s->rings[sel];
00359       struct io_uring_sqe *sqe = io_uring_get_sqe (ring);
00360       while (!sqe)
00361         {
00362           sqe = io_uring_get_sqe (ring);
```

```
00363           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00364         }
00365       if (!set_request (&oreq, &s->list, oreq->prev_remain, NULL, 0))
00366         {
00367           free (sqe);
00368           res = ERROR_CODE;
00369           continue;
00370         }
00371       io_uring_prep_readv (sqe, oreq->client_socket, &oreq->iov[0],
00372                            oreq->iovec_count, 0);
00373       io_uring_sqe_set_data (sqe, oreq);
00374       if (io_uring_submit (ring) < 0)
00375         {
00376           free (sqe);
00377           list_delete_node (&s->list, oreq);
00378           res = ERROR_CODE;
00379           continue;
00380         }
00381       res = SUCCESS_CODE;
00382     }
00383   pthread_mutex_unlock (&s->m_rings[sel]);
00384 }
00385
00386 // add_write
00387 static inline void
00388 add_write (struct saurion *const s, int fd, const char *const str,
00389            const int sel)
00390 {
00391   int res = ERROR_CODE;
00392   pthread_mutex_lock (&s->m_rings[sel]);
00393   while (res != SUCCESS_CODE)
00394     {
00395       struct io_uring *ring = &s->rings[sel];
00396       struct io_uring_sqe *sqe = io_uring_get_sqe (ring);
00397       while (!sqe)
00398         {
00399           sqe = io_uring_get_sqe (ring);
00400           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00401         }
00402       struct request *req = NULL;
00403       if (!set_request (&req, &s->list, strlen (str), (const void *const)str,
00404                         1))
00405         {
00406           free (sqe);
00407           res = ERROR_CODE;
00408           continue;
00409         }
00410       req->event_type = EV_WRI;
00411       req->client_socket = fd;
00412       io_uring_prep_writev (sqe, req->client_socket, req->iov,
00413                             req->iovec_count, 0);
00414       io_uring_sqe_set_data (sqe, req);
00415       if (io_uring_submit (ring) < 0)
00416         {
00417           free (sqe);
00418           list_delete_node (&s->list, req);
00419           res = ERROR_CODE;
00420           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00421           continue;
00422         }
00423       res = SUCCESS_CODE;
00424     }
00425   pthread_mutex_unlock (&s->m_rings[sel]);
00426 }
00427
00428 /****************** HANDLERS ******************/
00429 // handle_accept
00430 static inline void
00431 handle_accept (const struct saurion *const s, const int fd)
00432 {
00433   if (s->cb.on_connected)
00434     {
00435       s->cb.on_connected (fd, s->cb.on_connected_arg);
00436     }
00437 }
00438
00439 // calculate_max_iov_content
00440 [[nodiscard]]
00441 static inline size_t
00442 calculate_max_iov_content (const struct request *req)
00443 {
00444   size_t max_iov_cont = 0;
00445   for (size_t i = 0; i < req->iovec_count; ++i)
00446     {
00447       max_iov_cont += req->iov[i].iov_len;
00448     }
00449   return max_iov_cont;
```

```
00450 }
00451
00452 struct chunk_params
00453 {
00454   void **dest;
00455   void *dest_ptr;
00456   size_t dest_off;
00457   struct request *req;
00458   size_t cont_sz;
00459   size_t cont_rem;
00460   size_t max_iov_cont;
00461   size_t curr_iov;
00462   size_t curr_iov_off;
00463   size_t *len;
00464 };
00465
00466 // handle_previous_message
00467 [[nodiscard]]
00468 static inline int
00469 handle_previous_message (struct chunk_params *p)
00470 {
00471   p->cont_sz = p->req->prev_size;
00472   p->cont_rem = p->req->prev_remain;
00473   p->dest_off = p->cont_sz - p->cont_rem;
00474
00475   if (p->cont_rem <= p->max_iov_cont)
00476     {
00477       *p->dest = p->req->prev;
00478       p->dest_ptr = *p->dest;
00479       p->req->prev = NULL;
00480       p->req->prev_size = 0;
00481       p->req->prev_remain = 0;
00482     }
00483   else
00484     {
00485       p->dest_ptr = p->req->prev;
00486       *p->dest = NULL;
00487     }
00488   return SUCCESS_CODE;
00489 }
00490
00491 // handle_partial_message
00492 [[nodiscard]]
00493 static inline int
00494 handle_partial_message (struct chunk_params *p)
00495 {
00496   p->curr_iov = p->req->next_iov;
00497   p->curr_iov_off = p->req->next_offset;
00498
00499   p->cont_sz = *(size_t *)((uint8_t *)p->req->iov[p->curr_iov].iov_base
00500                            + p->curr_iov_off);
00501   p->cont_sz = ntohll (p->cont_sz);
00502   p->curr_iov_off += sizeof (uint64_t);
00503   p->cont_rem = p->cont_sz;
00504   p->dest_off = p->cont_sz - p->cont_rem;
00505
00506   if ((p->curr_iov_off + p->cont_rem + 1) <= p->max_iov_cont)
00507     {
00508       *p->dest = malloc (p->cont_sz);
00509       if (!*p->dest)
00510         {
00511           return ERROR_CODE;
00512         }
00513       p->dest_ptr = *p->dest;
00514     }
00515   else
00516     {
00517       p->req->prev = malloc (p->cont_sz);
00518       if (!p->req->prev)
00519         {
00520           return ERROR_CODE;
00521         }
00522       p->dest_ptr = p->req->prev;
00523       *p->dest = NULL;
00524       *p->len = 0;
00525     }
00526   return SUCCESS_CODE;
00527 }
00528
00529 // handle_new_message
00530 [[nodiscard]]
00531 static inline int
00532 handle_new_message (struct chunk_params *p)
00533 {
00534   p->curr_iov = 0;
00535   p->curr_iov_off = 0;
00536
```

```
00537   p->cont_sz = *(size_t *)((uint8_t *)p->req->iov[p->curr_iov].iov_base
00538                            + p->curr_iov_off);
00539   p->cont_sz = ntohll (p->cont_sz);
00540   p->curr_iov_off += sizeof (uint64_t);
00541   p->cont_rem = p->cont_sz;
00542   p->dest_off = p->cont_sz - p->cont_rem;
00543
00544   if (p->cont_rem <= p->max_iov_cont)
00545     {
00546       *p->dest = malloc (p->cont_sz);
00547       if (!*p->dest)
00548         {
00549           return ERROR_CODE; // Error al asignar memoria.
00550         }
00551       p->dest_ptr = *p->dest;
00552     }
00553   else
00554     {
00555       p->req->prev = malloc (p->cont_sz);
00556       if (!p->req->prev)
00557         {
00558           return ERROR_CODE; // Error al asignar memoria.
00559         }
00560       p->dest_ptr = p->req->prev;
00561       *p->dest = NULL;
00562     }
00563   return SUCCESS_CODE;
00564 }
00565
00566 // prepare_destination
00567 [[nodiscard]]
00568 static inline int
00569 prepare_destination (struct chunk_params *p)
00570 {
00571   if (p->req->prev && p->req->prev_size && p->req->prev_remain)
00572     {
00573       return handle_previous_message (p);
00574     }
00575   if (p->req->next_iov || p->req->next_offset)
00576     {
00577       return handle_partial_message (p);
00578     }
00579   return handle_new_message (p);
00580 }
00581
00582 // copy_data
00583 static inline void
00584 copy_data (struct chunk_params *p, uint8_t *ok)
00585 {
00586   size_t curr_iov_msg_rem = 0;
00587   *ok = 1UL;
00588   while (1)
00589     {
00590       curr_iov_msg_rem = MIN (
00591           p->cont_rem, (p->req->iov[p->curr_iov].iov_len - p->curr_iov_off));
00592       memcpy ((uint8_t *)p->dest_ptr + p->dest_off,
00593               (uint8_t *)p->req->iov[p->curr_iov].iov_base + p->curr_iov_off,
00594               curr_iov_msg_rem);
00595       p->dest_off += curr_iov_msg_rem;
00596       p->curr_iov_off += curr_iov_msg_rem;
00597       p->cont_rem -= curr_iov_msg_rem;
00598
00599       if (p->cont_rem <= 0)
00600         {
00601           if (*(((uint8_t *)p->req->iov[p->curr_iov].iov_base)
00602                 + p->curr_iov_off)
00603               != 0)
00604             {
00605               *ok = 0UL;
00606             }
00607           *p->len = p->cont_sz;
00608           ++p->curr_iov_off;
00609           break;
00610         }
00611       if (p->curr_iov_off >= (p->req->iov[p->curr_iov].iov_len))
00612         {
00613           ++p->curr_iov;
00614           if (p->curr_iov == p->req->iovec_count)
00615             {
00616               break;
00617             }
00618           p->curr_iov_off = 0;
00619         }
00620     }
00621 }
00622
00623 // validate_and_update
```

```
00624 [[nodiscard]]
00625 static inline uint8_t
00626 validate_and_update (struct chunk_params *p, uint8_t ok)
00627 {
00628   if (p->req->prev)
00629     {
00630       p->req->prev_size = p->cont_sz;
00631       p->req->prev_remain = p->cont_rem;
00632       *p->dest = NULL;
00633       *p->len = 0;
00634     }
00635   else
00636     {
00637       p->req->prev_size = 0;
00638       p->req->prev_remain = 0;
00639     }
00640   if (p->curr_iov < p->req->iovec_count)
00641     {
00642       uint64_t next_sz
00643           = *(uint64_t *)(((uint8_t *)p->req->iov[p->curr_iov].iov_base)
00644                           + p->curr_iov_off);
00645       if ((p->req->iov[p->curr_iov].iov_len > p->curr_iov_off) && next_sz)
00646         {
00647           p->req->next_iov = p->curr_iov;
00648           p->req->next_offset = p->curr_iov_off;
00649         }
00650       else
00651         {
00652           p->req->next_iov = 0;
00653           p->req->next_offset = 0;
00654         }
00655     }
00656
00657   return ok ?  SUCCESS_CODE : ERROR_CODE;
00658 }
00659
00660 // read_chunk_free
00661 static inline void
00662 read_chunk_free (struct chunk_params *p)
00663 {
00664   free (p->dest_ptr);
00665   p->dest_ptr = NULL;
00666   *p->dest = NULL;
00667   *p->len = 0;
00668   p->req->next_iov = 0;
00669   p->req->next_offset = 0;
00670   for (size_t i = p->curr_iov; i < p->req->iovec_count; ++i)
00671     {
00672       for (size_t j = p->curr_iov_off; j < p->req->iov[i].iov_len; ++j)
00673         {
00674           uint8_t foot = *((uint8_t *)p->req->iov[i].iov_base) + j;
00675           if (foot == 0)
00676             {
00677               p->req->next_iov = i;
00678               p->req->next_offset = (j + 1) % p->req->iov[i].iov_len;
00679               return;
00680             }
00681         }
00682     }
00683 }
00684
00685 // read_chunk
00686 [[nodiscard]]
00687 int
00688 read_chunk (void **dest, size_t *len, struct request *const req)
00689 {
00690   struct chunk_params p;
00691   p.req = req;
00692   p.dest = dest;
00693   p.len = len;
00694   if (p.req->iovec_count == 0)
00695     {
00696       return ERROR_CODE;
00697     }
00698
00699   p.max_iov_cont = calculate_max_iov_content (p.req);
00700   p.cont_sz = 0;
00701   p.cont_rem = 0;
00702   p.curr_iov = 0;
00703   p.curr_iov_off = 0;
00704   p.dest_off = 0;
00705   p.dest_ptr = NULL;
00706   if (!prepare_destination (&p))
00707     {
00708       return ERROR_CODE;
00709     }
00710
```

```
00711   uint8_t ok = 1UL;
00712   copy_data (&p, &ok);
00713
00714   if (validate_and_update (&p, ok))
00715     {
00716       return SUCCESS_CODE;
00717     }
00718   read_chunk_free (&p);
00719   return ERROR_CODE;
00720 }
00721
00722 // handle_read
00723 static inline void
00724 handle_read (struct saurion *const s, struct request *const req)
00725 {
00726   void *msg = NULL;
00727   size_t len = 0;
00728   while (1)
00729     {
00730       if (!read_chunk (&msg, &len, req))
00731         {
00732           break;
00733         }
00734       if (req->next_iov || req->next_offset)
00735         {
00736           if (s->cb.on_readed && msg)
00737             {
00738               s->cb.on_readed (req->client_socket, msg, len,
00739                                s->cb.on_readed_arg);
00740             }
00741           free (msg);
00742           msg = NULL;
00743           continue;
00744         }
00745       if (req->prev && req->prev_size && req->prev_remain)
00746         {
00747           add_read_continue (s, req, next (s));
00748           return;
00749         }
00750       if (s->cb.on_readed && msg)
00751         {
00752           s->cb.on_readed (req->client_socket, msg, len, s->cb.on_readed_arg);
00753         }
00754       free (msg);
00755       msg = NULL;
00756       break;
00757     }
00758   add_read (s, req->client_socket);
00759 }
00760
00761 // handle_write
00762 static inline void
00763 handle_write (const struct saurion *const s, const int fd)
00764 {
00765   if (s->cb.on_wrote)
00766     {
00767       s->cb.on_wrote (fd, s->cb.on_wrote_arg);
00768     }
00769 }
00770
00771 // handle_error
00772 static inline void
00773 handle_error (const struct saurion *const s, const struct request *const req)
00774 {
00775   if (s->cb.on_error)
00776     {
00777       const char *resp = "ERROR";
00778       s->cb.on_error (req->client_socket, resp, (ssize_t)strlen (resp),
00779                       s->cb.on_error_arg);
00780     }
00781 }
00782
00783 // handle_close
00784 static inline void
00785 handle_close (const struct saurion *const s, const struct request *const req)
00786 {
00787   if (s->cb.on_closed)
00788     {
00789       s->cb.on_closed (req->client_socket, s->cb.on_closed_arg);
00790     }
00791   close (req->client_socket);
00792 }
00793
00794 /****************** INTERFACE ******************/
00795 // saurion_set_socket
00796 [[nodiscard]] int
00797 saurion_set_socket (const int p)
```

```
00798 {
00799   int sock = 0;
00800   struct sockaddr_in srv_addr;
00801
00802   sock = socket (PF_INET, SOCK_STREAM, 0);
00803   if (sock < 1)
00804     {
00805       return ERROR_CODE;
00806     }
00807
00808   int enable = 1;
00809   if (setsockopt (sock, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof (int)) < 0)
00810     {
00811       return ERROR_CODE;
00812     }
00813
00814   memset (&srv_addr, 0, sizeof (srv_addr));
00815   srv_addr.sin_family = AF_INET;
00816   srv_addr.sin_port = htons (p);
00817   srv_addr.sin_addr.s_addr = htonl (INADDR_ANY);
00818
00819   if (bind (sock, (const struct sockaddr *)&srv_addr, sizeof (srv_addr)) < 0)
00820     {
00821       return ERROR_CODE;
00822     }
00823
00824   if (listen (sock, ACCEPT_QUEUE) < 0)
00825     {
00826       return ERROR_CODE;
00827     }
00828
00829   return sock;
00830 }
00831
00832 // saurion_create
00833 [[nodiscard]]
00834 struct saurion *
00835 saurion_create (uint32_t n_threads)
00836 {
00837   LOG_INIT (" ");
00838   struct saurion *p = (struct saurion *)malloc (sizeof (struct saurion));
00839   if (!p)
00840     {
00841       LOG_END (" ");
00842       return NULL;
00843     }
00844   int ret = 0;
00845   ret = pthread_mutex_init (&p->status_m, NULL);
00846   if (ret)
00847     {
00848       free (p);
00849       LOG_END (" ");
00850       return NULL;
00851     }
00852   ret = pthread_cond_init (&p->status_c, NULL);
00853   if (ret)
00854     {
00855       free (p);
00856       LOG_END (" ");
00857       return NULL;
00858     }
00859   p->m_rings
00860       = (pthread_mutex_t *)malloc (n_threads * sizeof (pthread_mutex_t));
00861   if (!p->m_rings)
00862     {
00863       free (p);
00864       LOG_END (" ");
00865       return NULL;
00866     }
00867   for (uint32_t i = 0; i < n_threads; ++i)
00868     {
00869       pthread_mutex_init (&(p->m_rings[i]), NULL);
00870     }
00871   p->ss = 0;
00872   n_threads = (n_threads < 2 ?  2 :  n_threads);
00873   n_threads = (n_threads > NUM_CORES ? NUM_CORES : n_threads);
00874   p->n_threads = n_threads;
00875   p->status = 0;
00876   p->list = NULL;
00877   p->cb.on_connected = NULL;
00878   p->cb.on_connected_arg = NULL;
00879   p->cb.on_readed = NULL;
00880   p->cb.on_readed_arg = NULL;
00881   p->cb.on_wrote = NULL;
00882   p->cb.on_wrote_arg = NULL;
00883   p->cb.on_closed = NULL;
00884   p->cb.on_closed_arg = NULL;
```

```
00885   p->cb.on_error = NULL;
00886   p->cb.on_error_arg = NULL;
00887   p->next = 0;
00888   p->efds = (int *)malloc (sizeof (int) * p->n_threads);
00889   if (!p->efds)
00890     {
00891       free (p->m_rings);
00892       free (p);
00893       LOG_END (" ");
00894       return NULL;
00895     }
00896   for (uint32_t i = 0; i < p->n_threads; ++i)
00897     {
00898       p->efds[i] = eventfd (0, EFD_NONBLOCK);
00899       if (p->efds[i] == ERROR_CODE)
00900         {
00901           for (uint32_t j = 0; j < i; ++j)
00902             {
00903               close (p->efds[j]);
00904             }
00905           free (p->efds);
00906           free (p->m_rings);
00907           free (p);
00908           LOG_END (" ");
00909           return NULL;
00910         }
00911     }
00912   p->rings
00913       = (struct io_uring *)malloc (sizeof (struct io_uring) * p->n_threads);
00914   if (!p->rings)
00915     {
00916       for (uint32_t j = 0; j < p->n_threads; ++j)
00917         {
00918           close (p->efds[j]);
00919         }
00920       free (p->efds);
00921       free (p->m_rings);
00922       free (p);
00923       LOG_END (" ");
00924       return NULL;
00925     }
00926   for (uint32_t i = 0; i < p->n_threads; ++i)
00927     {
00928       memset (&p->rings[i], 0, sizeof (struct io_uring));
00929       ret = io_uring_queue_init (SAURION_RING_SIZE, &p->rings[i], 0);
00930       if (ret)
00931         {
00932           for (uint32_t j = 0; j < p->n_threads; ++j)
00933             {
00934               close (p->efds[j]);
00935             }
00936           free (p->efds);
00937           free (p->rings);
00938           free (p->m_rings);
00939           free (p);
00940           LOG_END (" ");
00941           return NULL;
00942         }
00943     }
00944   p->pool = threadpool_create (p->n_threads);
00945   LOG_END (" ");
00946   return p;
00947 }
00948
00949 // handle_event_read
00950 static inline void
00951 handle_event_read (const struct io_uring_cqe *const cqe,
00952                    struct saurion *const s, struct request *req)
00953 {
00954   if (cqe->res < 0)
00955     {
00956       handle_error (s, req);
00957     }
00958   if (cqe->res < 1)
00959     {
00960       handle_close (s, req);
00961     }
00962   if (cqe->res > 0)
00963     {
00964       handle_read (s, req);
00965     }
00966   list_delete_node (&s->list, req);
00967 }
00968
00969 // saurion_worker_master_loop_it
00970 [[nodiscard]]
00971 static inline int
```

```
00972 saurion_worker_master_loop_it (struct saurion *const s,
00973                                 struct sockaddr_in *client_addr,
00974                                 socklen_t *client_addr_len)
00975 {
00976   LOG_INIT (" ");
00977   struct io_uring ring = s->rings[0];
00978   struct io_uring_cqe *cqe = NULL;
00979   int ret = io_uring_wait_cqe (&ring, &cqe);
00980   if (ret < 0)
00981     {
00982       free (cqe);
00983       LOG_END (" ");
00984       return CRITICAL_CODE;
00985     }
00986   struct request *req = (struct request *)cqe->user_data;
00987   if (!req)
00988     {
00989       io_uring_cqe_seen (&s->rings[0], cqe);
00990       LOG_END (" ");
00991       return SUCCESS_CODE;
00992     }
00993   if (cqe->res < 0)
00994     {
00995       list_delete_node (&s->list, req);
00996       LOG_END (" ");
00997       return CRITICAL_CODE;
00998     }
00999   if (req->client_socket == s->efds[0])
01000     {
01001       io_uring_cqe_seen (&s->rings[0], cqe);
01002       list_delete_node (&s->list, req);
01003       LOG_END (" ");
01004       return ERROR_CODE;
01005     }
01006   io_uring_cqe_seen (&s->rings[0], cqe);
01007   switch (req->event_type)
01008     {
01009     case EV_ACC:
01010       handle_accept (s, cqe->res);
01011       add_accept (s, client_addr, client_addr_len);
01012       add_read (s, cqe->res);
01013       list_delete_node (&s->list, req);
01014       break;
01015     case EV_REA:
01016       handle_event_read (cqe, s, req);
01017       break;
01018     case EV_WRI:
01019       handle_write (s, req->client_socket);
01020       list_delete_node (&s->list, req);
01021       break;
01022     }
01023   LOG_END (" ");
01024   return SUCCESS_CODE;
01025 }
01026
01027 // saurion_worker_master
01028 void
01029 saurion_worker_master (void *arg)
01030 {
01031   LOG_INIT (" ");
01032   struct saurion *const s = (struct saurion *)arg;
01033   struct sockaddr_in client_addr;
01034   socklen_t client_addr_len = sizeof (client_addr);
01035
01036   add_efd (s, s->efds[0], 0);
01037   add_accept (s, &client_addr, &client_addr_len);
01038
01039   pthread_mutex_lock (&s->status_m);
01040   ++s->status;
01041   pthread_cond_broadcast (&s->status_c);
01042   pthread_mutex_unlock (&s->status_m);
01043   while (1)
01044     {
01045       int ret
01046           = saurion_worker_master_loop_it (s, &client_addr, &client_addr_len);
01047       if (ret == ERROR_CODE || ret == CRITICAL_CODE)
01048         {
01049           break;
01050         }
01051     }
01052   pthread_mutex_lock (&s->status_m);
01053   --s->status;
01054   pthread_cond_signal (&s->status_c);
01055   pthread_mutex_unlock (&s->status_m);
01056   LOG_END (" ");
01057   return;
01058 }
```

```
01059
01060 // saurion_worker_slave_loop_it
01061 [[nodiscard]]
01062 static inline int
01063 saurion_worker_slave_loop_it (struct saurion *const s, const int sel)
01064 {
01065   LOG_INIT (" ");
01066   struct io_uring ring = s->rings[sel];
01067   struct io_uring_cqe *cqe = NULL;
01068
01069   add_efd (s, s->efds[sel], sel);
01070   int ret = io_uring_wait_cqe (&ring, &cqe);
01071   if (ret < 0)
01072     {
01073       free (cqe);
01074       LOG_END (" ");
01075       return CRITICAL_CODE;
01076     }
01077   struct request *req = (struct request *)cqe->user_data;
01078   if (!req)
01079     {
01080       io_uring_cqe_seen (&ring, cqe);
01081       LOG_END (" ");
01082       return SUCCESS_CODE;
01083     }
01084   if (cqe->res < 0)
01085     {
01086       list_delete_node (&s->list, req);
01087       LOG_END (" ");
01088       return CRITICAL_CODE;
01089     }
01090   if (req->client_socket == s->efds[sel])
01091     {
01092       io_uring_cqe_seen (&ring, cqe);
01093       list_delete_node (&s->list, req);
01094       LOG_END (" ");
01095       return ERROR_CODE;
01096     }
01097   io_uring_cqe_seen (&ring, cqe);
01098   switch (req->event_type)
01099     {
01100     case EV_REA:
01101       handle_event_read (cqe, s, req);
01102       break;
01103     case EV_WRI:
01104       handle_write (s, req->client_socket);
01105       list_delete_node (&s->list, req);
01106       break;
01107     }
01108   LOG_END (" ");
01109   return SUCCESS_CODE;
01110 }
01111
01112 // saurion_worker_slave
01113 void
01114 saurion_worker_slave (void *arg)
01115 {
01116   LOG_INIT (" ");
01117   struct saurion_wrapper *const ss = (struct saurion_wrapper *)arg;
01118   struct saurion *s = ss->s;
01119   const int sel = ss->sel;
01120   free (ss);
01121
01122   add_efd (s, s->efds[sel], sel);
01123
01124   pthread_mutex_lock (&s->status_m);
01125   ++s->status;
01126   pthread_cond_broadcast (&s->status_c);
01127   pthread_mutex_unlock (&s->status_m);
01128   while (1)
01129     {
01130       int res = saurion_worker_slave_loop_it (s, sel);
01131       if (res == ERROR_CODE || res == CRITICAL_CODE)
01132         {
01133           break;
01134         }
01135     }
01136   pthread_mutex_lock (&s->status_m);
01137   --s->status;
01138   pthread_cond_signal (&s->status_c);
01139   pthread_mutex_unlock (&s->status_m);
01140   LOG_END (" ");
01141   return;
01142 }
01143
01144 // saurion_start
01145 [[nodiscard]]
```

```
01146 int
01147 saurion_start (struct saurion *const s)
01148 {
01149   threadpool_init (s->pool);
01150   threadpool_add (s->pool, saurion_worker_master, s);
01151   struct saurion_wrapper *ss = NULL;
01152   for (uint32_t i = 1; i < s->n_threads; ++i)
01153     {
01154       ss = (struct saurion_wrapper *)malloc (sizeof (struct saurion_wrapper));
01155       if (!ss)
01156         {
01157           return ERROR_CODE;
01158         }
01159       ss->s = s;
01160       ss->sel = i;
01161       threadpool_add (s->pool, saurion_worker_slave, ss);
01162     }
01163   pthread_mutex_lock (&s->status_m);
01164   while (s->status < (int)s->n_threads)
01165     {
01166       pthread_cond_wait (&s->status_c, &s->status_m);
01167     }
01168   pthread_mutex_unlock (&s->status_m);
01169   return SUCCESS_CODE;
01170 }
01171
01172 // saurion_stop
01173 void
01174 saurion_stop (const struct saurion *const s)
01175 {
01176   uint64_t u = 1;
01177   for (uint32_t i = 0; i < s->n_threads; ++i)
01178     {
01179       while (write (s->efds[i], &u, sizeof (u)) < 0)
01180         {
01181           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
01182         }
01183     }
01184   threadpool_wait_empty (s->pool);
01185 }
01186
01187 // saurion_destroy
01188 void
01189 saurion_destroy (struct saurion *const s)
01190 {
01191   pthread_mutex_lock (&s->status_m);
01192   while (s->status > 0)
01193     {
01194       pthread_cond_wait (&s->status_c, &s->status_m);
01195     }
01196   pthread_mutex_unlock (&s->status_m);
01197   threadpool_destroy (s->pool);
01198   for (uint32_t i = 0; i < s->n_threads; ++i)
01199     {
01200       io_uring_queue_exit (&s->rings[i]);
01201       pthread_mutex_destroy (&s->m_rings[i]);
01202     }
01203   free (s->m_rings);
01204   list_free (&s->list);
01205   for (uint32_t i = 0; i < s->n_threads; ++i)
01206     {
01207       close (s->efds[i]);
01208     }
01209   free (s->efds);
01210   if (!s->ss)
01211     {
01212       close (s->ss);
01213     }
01214   free (s->rings);
01215   pthread_mutex_destroy (&s->status_m);
01216   pthread_cond_destroy (&s->status_c);
01217   free (s);
01218 }
01219
01220 // saurion_send
01221 void
01222 saurion_send (struct saurion *const s, const int fd, const char *const msg)
01223 {
01224   add_write (s, fd, msg, next (s));
01225 }
```

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

```
#include <pthread.h>
#include <stdio.h>
```

Include dependency graph for main.c:

7.16 main.c

Go to the documentation of this file.

```
00001 #include <pthread.h> // for pthread_create, pthread_join, pthread_t
00002 #include <stdio.h>   // for printf, fprintf, NULL, stderr
00003
00004 int counter = 0;
00005
00006 void *
00007 increment (void *arg)
00008 {
00009   int id = *((int *)arg);
00010   for (int i = 0; i < 100000; ++i)
00011     {
00012       counter++;
00013       if (i % 10000 == 0)
00014         {
00015           printf ("Thread %d at iteration %d\n", id, i);
00016         }
00017     }
00018   printf ("Thread %d finished\n", id);
00019   return NULL;
00020 }
00021
00022 int
00023 main ()
00024 {
00025   pthread_t t1;
00026   pthread_t t2;
00027   int id1 = 1;
00028   int id2 = 2;
00029
00030   printf ("Starting threads...\n");
00031
00032   if (pthread_create (&t1, NULL, increment, &id1))
00033     {
00034       fprintf (stderr, "Error creating thread 1\n");
00035       return 1;
00036     }
00037   if (pthread_create (&t2, NULL, increment, &id2))
00038     {
00039       fprintf (stderr, "Error creating thread 2\n");
00040       return 1;
00041     }
00042
00043   printf ("Waiting for thread 1 to join...\n");
00044   if (pthread_join (t1, NULL))
00045     {
00046       fprintf (stderr, "Error joining thread 1\n");
00047       return 2;
00048     }
00049   printf ("Thread 1 joined\n");
00050
00051   printf ("Waiting for thread 2 to join...\n");
00052   if (pthread_join (t2, NULL))
00053     {
00054       fprintf (stderr, "Error joining thread 2\n");
00055       return 2;
00056     }
00057   printf ("Thread 2 joined\n");
00058
00059   printf ("Final counter value:  %d\n", counter);
00060   return 0;
00061 }
```

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

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

Include dependency graph for saurion.cpp:

7.18 saurion.cpp

Go to the documentation of this file.

```
00001 #include "saurion.hpp"
00002
00003 #include "low_saurion.h" // for saurion, saurion_create, saurion_destroy
00004
00005 Saurion::Saurion (const uint32_t thds, const int sck) noexcept
00006 {
00007   this->s = saurion_create (thds);
00008   if (!this->s)
00009     {
00010       return;
00011     }
00012   this->s->ss = sck;
00013 }
00014
00015 Saurion::~Saurion () { saurion_destroy (this->s); }
00016
00017 void
00018 Saurion::init () noexcept
00019 {
00020   if (!saurion_start (this->s))
00021     {
00022       return;
00023     }
00024 }
00025
00026 void
00027 Saurion::stop () const noexcept
00028 {
00029   saurion_stop (this->s);
00030 }
00031
00032 Saurion *
00033 Saurion::on_connected (Saurion::ConnectedCb ncb, void *arg) noexcept
00034 {
00035   s->cb.on_connected = ncb;
00036   s->cb.on_connected_arg = arg;
00037   return this;
00038 }
00039
00040 Saurion *
00041 Saurion::on_readed (Saurion::ReadedCb ncb, void *arg) noexcept
00042 {
00043   s->cb.on_readed = ncb;
00044   s->cb.on_readed_arg = arg;
00045   return this;
00046 }
00047
00048 Saurion *
00049 Saurion::on_wrote (Saurion::WroteCb ncb, void *arg) noexcept
00050 {
00051   s->cb.on_wrote = ncb;
00052   s->cb.on_wrote_arg = arg;
00053   return this;
00054 }
00055
00056 Saurion *
00057 Saurion::on_closed (Saurion::ClosedCb ncb, void *arg) noexcept
00058 {
00059   s->cb.on_closed = ncb;
00060   s->cb.on_closed_arg = arg;
00061   return this;
00062 }
00063
00064 Saurion *
00065 Saurion::on_error (Saurion::ErrorCb ncb, void *arg) noexcept
00066 {
00067   s->cb.on_error = ncb;
00068   s->cb.on_error_arg = arg;
00069   return this;
00070 }
00071
00072 void
00073 Saurion::send (const int fd, const char *const msg) noexcept
00074 {
00075   saurion_send (this->s, fd, msg);
00076 }
```

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

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

Include dependency graph for threadpool.c:

Classes

- struct task
- struct threadpool

Macros

- #define TRUE 1
- #define FALSE 0

Functions

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

7.19.1 Macro Definition Documentation

7.19.1.1 FALSE

```
#define FALSE 0
```

Definition at line 9 of file threadpool.c.

7.19.1.2 TRUE

```
#define TRUE 1
```

Definition at line 8 of file threadpool.c.

7.19.2 Function Documentation

7.19.2.1 threadpool_worker()

```
void * threadpool_worker (
            void * arg )
```

Definition at line 107 of file threadpool.c.

```
00108 {
00109   LOG_INIT (" ");
00110   struct threadpool *pool = (struct threadpool *)arg;
00111   while (TRUE)
00112     {
00113       pthread_mutex_lock (&pool->queue_lock);
00114       while (pool->task_queue_head == NULL && !pool->stop)
00115         {
00116           pthread_cond_wait (&pool->queue_cond, &pool->queue_lock);
00117         }
00118
00119       if (pool->stop && pool->task_queue_head == NULL)
00120         {
00121           pthread_mutex_unlock (&pool->queue_lock);
00122           break;
00123         }
00124
00125       struct task *task = pool->task_queue_head;
00126       if (task != NULL)
00127         {
00128           pool->task_queue_head = task->next;
00129           if (pool->task_queue_head == NULL)
00130             pool->task_queue_tail = NULL;
00131
00132           if (pool->task_queue_head == NULL)
00133             {
00134               pthread_cond_signal (&pool->empty_cond);
00135             }
00136         }
00137       pthread_mutex_unlock (&pool->queue_lock);
00138
00139       if (task != NULL)
00140         {
00141           task->function (task->argument);
00142           free (task);
00143         }
00144     }
00145   LOG_END (" ");
00146   pthread_exit (NULL);
00147   return NULL;
00148 }
```

7.20 threadpool.c

Go to the documentation of this file.

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

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

```
00106 void *
00107 threadpool_worker (void *arg)
00108 {
00109   LOG_INIT (" ");
00110   struct threadpool *pool = (struct threadpool *)arg;
00111   while (TRUE)
00112     {
00113       pthread_mutex_lock (&pool->queue_lock);
00114       while (pool->task_queue_head == NULL && !pool->stop)
00115         {
00116           pthread_cond_wait (&pool->queue_cond, &pool->queue_lock);
00117         }
00118
00119       if (pool->stop && pool->task_queue_head == NULL)
00120         {
00121           pthread_mutex_unlock (&pool->queue_lock);
00122           break;
00123         }
00124
00125       struct task *task = pool->task_queue_head;
00126       if (task != NULL)
00127         {
00128           pool->task_queue_head = task->next;
00129           if (pool->task_queue_head == NULL)
00130             pool->task_queue_tail = NULL;
00131
00132           if (pool->task_queue_head == NULL)
00133             {
00134               pthread_cond_signal (&pool->empty_cond);
00135             }
00136         }
00137       pthread_mutex_unlock (&pool->queue_lock);
00138
00139       if (task != NULL)
00140         {
00141           task->function (task->argument);
00142           free (task);
00143         }
00144     }
00145   LOG_END (" ");
00146   pthread_exit (NULL);
00147   return NULL;
00148 }
00149
00150 void
00151 threadpool_init (struct threadpool *pool)
00152 {
00153   LOG_INIT (" ");
00154   if (pool == NULL || pool->started)
00155     {
00156       LOG_END (" ");
00157       return;
00158     }
00159   for (size_t i = 0; i < pool->num_threads; i++)
00160     {
00161       if (pthread_create (&pool->threads[i], NULL, threadpool_worker,
00162                           (void *)pool)
00163           != 0)
00164         {
00165           perror ("Failed to create thread");
00166           pool->stop = TRUE;
00167           break;
00168         }
00169     }
00170   pool->started = TRUE;
00171   LOG_END (" ");
00172 }
00173
00174 void
00175 threadpool_add (struct threadpool *pool, void (*function) (void *),
00176                 void *argument)
00177 {
00178   LOG_INIT (" ");
00179   if (pool == NULL || function == NULL)
00180     {
00181       LOG_END (" ");
00182       return;
00183     }
00184
00185   struct task *new_task = malloc (sizeof (struct task));
00186   if (new_task == NULL)
00187     {
00188       perror ("Failed to allocate task");
00189       LOG_END (" ");
00190       return;
00191     }
00192
```

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

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

Index