Saurion

Generated by Doxygen 1.9.4

Chapter 1

Todo List

Member EXTERNAL_set_socket (int p)

Eliminar

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

add message contraint

validar `msg_size`, crear maximos

validar `offsets`

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 PACKING_SZ 128

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

Functions

- int EXTERNAL_set_socket (int p)
- struct saurion ∗ saurion_create (uint32_t n_threads)

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

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

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

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

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

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

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

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

Author

Israel

Date

2024

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

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

Parameters

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

Return values

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

Note

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

5.1.2 Macro Definition Documentation

5.1.2.1 PACKING_SZ

```
#define PACKING_SZ 128
```

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

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

In this case, the value is set to 128 bytes, meaning that structures marked with `__attribute__((aligned(↩ PACKING_SZ)))` will be aligned to 128-byte boundaries.

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

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

Definition at line 129 of file low_saurion.h.

5.1.3 Function Documentation

5.1.3.1 allocate_iovec()

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

Definition at line 114 of file low_saurion.c.

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

5.1.3.2 EXTERNAL_set_socket()

```
int EXTERNAL_set_socket (
              int p )
```

Todo Eliminar

Definition at line 548 of file low_saurion.c.

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

5.1.3.3 free_request()

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

Definition at line 61 of file low_saurion.c.

```
00061                                                                            {
00062   if (children_ptr) {
00063     free(children_ptr);
00064     children_ptr = NULL;
00065   }
00066   for (size_t i = 0; i < amount; ++i) {
00067     free(req->iov[i].iov_base);
00068     req->iov[i].iov_base = NULL;
00069   }
00070   free(req);
00071   req = NULL;
00072   free(children_ptr);
00073   children_ptr = NULL;
00074 }
```

5.1.3.4 initialize_iovec()

```
int initialize_iovec (
            struct iovec * iov,
            size_t amount,
            size_t pos,
            const void * msg,
            size_t size,
            uint8_t h )  [private]
```

Initializes a specified `iovec` structure with a message fragment.

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

Parameters

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

Return values

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

Note

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

Attention

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

Warning

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

Definition at line 77 of file low_saurion.c.

```
                                     {
  if (!iov || !iov->iov_base) {
    return ERROR_CODE;
  }
  if (msg) {
    size_t len = iov->iov_len;
    char *dest = (char *)iov->iov_base;
    char *orig = (char *)msg + pos * CHUNK_SZ;
    size_t cpy_sz = 0;
    if (h) {
      if (pos == 0) {
        uint64_t send_size = htonll(size);
        memcpy(dest, &send_size, sizeof(uint64_t));
        dest += sizeof(uint64_t);
        len -= sizeof(uint64_t);
      } else {
        orig -= sizeof(uint64_t);
      }
      if ((pos + 1) == amount) {
        --len;
        cpy_sz = (len < size ? len : size);
        dest[cpy_sz] = 0;
      }
    }
    cpy_sz = (len < size ? len : size);
    memcpy(dest, orig, cpy_sz);
    dest += cpy_sz;
    size_t rem = CHUNK_SZ - (dest - (char *)iov->iov_base);
    memset(dest, 0, rem);
  } else {
    memset((char *)iov->iov_base, 0, CHUNK_SZ);
  }
  return SUCCESS_CODE;
}
```

5.1.3.5 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`

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

Definition at line 350 of file low_saurion.c.

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

```
      *dest = NULL;
      *len = 0;
    }
  } else {
    // Reading the first message
    curr_iov = 0;
    curr_iov_off = 0;
    cont_sz = *((size_t *)(req->iov[curr_iov].iov_base + curr_iov_off));
    cont_sz = ntohll(cont_sz);
    curr_iov_off += sizeof(uint64_t);
    cont_rem = cont_sz;
    dest_off = cont_sz - cont_rem;
    if (cont_rem <= max_iov_cont) {
      *dest = malloc(cont_sz);
      dest_ptr = *dest;
    } else {
      req->prev = malloc(cont_sz);
      dest_ptr = req->prev;
      *dest = NULL;
    }
  }
  size_t curr_iov_msg_rem = 0;

  // Copy loop
  uint8_t ok = 1UL;
  while (1) {
    curr_iov_msg_rem = MIN(cont_rem, (req->iov[curr_iov].iov_len - curr_iov_off));
    memcpy(dest_ptr + dest_off, req->iov[curr_iov].iov_base + curr_iov_off, curr_iov_msg_rem);
    dest_off += curr_iov_msg_rem;
    curr_iov_off += curr_iov_msg_rem;
    cont_rem -= curr_iov_msg_rem;
    if (cont_rem <= 0) {
      // Finish reading
      if (*((uint8_t *)(req->iov[curr_iov].iov_base + curr_iov_off)) != 0) {
        ok = 0UL;
      }
      *len = cont_sz;
      ++curr_iov_off;
      break;
    }
    if (curr_iov_off >= (req->iov[curr_iov].iov_len)) {
      ++curr_iov;
      if (curr_iov == req->iovec_count) {
        break;
      }
      curr_iov_off = 0;
    }
  }

  // Update status
  if (req->prev) {
    req->prev_size = cont_sz;
    req->prev_remain = cont_rem;
    *dest = NULL;
    len = 0;
  } else {
    req->prev_size = 0;
    req->prev_remain = 0;
  }
  if (curr_iov < req->iovec_count) {
    uint64_t next_sz = *(uint64_t *)(req->iov[curr_iov].iov_base + curr_iov_off);
    if ((req->iov[curr_iov].iov_len > curr_iov_off) && next_sz) {
      req->next_iov = curr_iov;
      req->next_offset = curr_iov_off;
    } else {
      req->next_iov = 0;
      req->next_offset = 0;
    }
  }

  // Finish
  if (!ok) {
    // Esto solo es posible si no se encuentra un 0 al final del la lectura
    // buscar el siguiente 0 y ...  probar fortuna
    free(dest_ptr);
    dest_ptr = NULL;
    *dest = NULL;
    *len = 0;
    req->next_iov = 0;
    req->next_offset = 0;
    for (size_t i = curr_iov; i < req->iovec_count; ++i) {
      for (size_t j = curr_iov_off; j < req->iov[i].iov_len; ++j) {
        uint8_t foot = *(uint8_t *)(req->iov[i].iov_base + j);
        if (foot == 0) {
          req->next_iov = i;
          req->next_offset = (j + 1) % req->iov[i].iov_len;
          return ERROR_CODE;
```

```
00487         }
00488       }
00489     }
00490     return ERROR_CODE;
00491   }
00492   return SUCCESS_CODE;
00493 }
```

5.1.3.6 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 585 of file low_saurion.c.

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

```
  p->cb.on_error = NULL;
  p->cb.on_error_arg = NULL;
  p->next = 0;
  // Inicializar efds
  p->efds = (int *)malloc(sizeof(int) * p->n_threads);
  if (!p->efds) {
    free(p->m_rings);
    free(p);
    return NULL;
  }
  for (uint32_t i = 0; i < p->n_threads; ++i) {
    p->efds[i] = eventfd(0, EFD_NONBLOCK);
    if (p->efds[i] == ERROR_CODE) {
      for (uint32_t j = 0; j < i; ++j) {
        close(p->efds[j]);
      }
      free(p->efds);
      free(p->m_rings);
      free(p);
      return NULL;
    }
  }
  // Inicializar rings
  p->rings = (struct io_uring *)malloc(sizeof(struct io_uring) * p->n_threads);
  if (!p->rings) {
    for (uint32_t j = 0; j < p->n_threads; ++j) {
      close(p->efds[j]);
    }
    free(p->efds);
    free(p->m_rings);
    free(p);
    return NULL;
  }
  for (uint32_t i = 0; i < p->n_threads; ++i) {
    memset(&p->rings[i], 0, sizeof(struct io_uring));
    ret = io_uring_queue_init(SAURION_RING_SIZE, &p->rings[i], 0);
    if (ret) {
      for (uint32_t j = 0; j < p->n_threads; ++j) {
        close(p->efds[j]);
      }
      free(p->efds);
      free(p->rings);
      free(p->m_rings);
      free(p);
      return NULL;
    }
  }
  p->pool = ThreadPool_create(p->n_threads);
  return p;
}
```

5.1.3.7 saurion_destroy()

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

Destroys the saurion structure and frees all associated resources.

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

Parameters

s	Pointer to the saurion structure.

Definition at line 832 of file low_saurion.c.

```
00832                                               {
00833   pthread_mutex_lock(&s->status_m);
00834   while (s->status == 1) {
00835     pthread_cond_wait(&s->status_c, &s->status_m);
00836   }
00837   pthread_mutex_unlock(&s->status_m);
00838   ThreadPool_destroy(s->pool);
00839   for (uint32_t i = 0; i < s->n_threads; ++i) {
00840     io_uring_queue_exit(&s->rings[i]);
00841     pthread_mutex_destroy(&s->m_rings[i]);
00842   }
00843   free(s->m_rings);
00844   list_free(&s->list);
00845   for (uint32_t i = 0; i < s->n_threads; ++i) {
00846     close(s->efds[i]);
00847   }
00848   free(s->efds);
00849   if (!s->ss) {
00850     close(s->ss);
00851   }
00852   free(s->rings);
00853   pthread_mutex_destroy(&s->status_m);
00854   pthread_cond_destroy(&s->status_c);
00855   free(s);
00856 }
```

5.1.3.8 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 858 of file low_saurion.c.

```
00858                                                                         {
00859   add_write(s, fd, msg, next(s));
00860 }
```

5.1.3.9 saurion_start()

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

Starts event processing in the `saurion` structure.

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

Parameters

s	Pointer to the `saurion` structure.

Returns

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

Definition at line 808 of file low_saurion.c.

```
00808                                               {
00809   pthread_mutex_init(&print_mutex, NULL);
00810   ThreadPool_init(s->pool);
00811   ThreadPool_add_default(s->pool, saurion_worker_master, s);
00812   struct saurion_wrapper *ss = NULL;
00813   for (uint32_t i = 1; i < s->n_threads; ++i) {
00814     ss = (struct saurion_wrapper *)malloc(sizeof(struct saurion_wrapper));
00815     ss->s = s;
00816     ss->sel = i;
00817     ThreadPool_add_default(s->pool, saurion_worker_slave, ss);
00818   }
00819   return SUCCESS_CODE;
00820 }
```

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 822 of file low_saurion.c.

```
00822                                                     {
00823   uint64_t u = 1;
00824   for (uint32_t i = 0; i < s->n_threads; ++i) {
00825     while (write(s->efds[i], &u, sizeof(u)) < 0) {
00826       usleep(TIMEOUT_RETRY);
00827     }
00828   }
00829   ThreadPool_wait_empty(s->pool);
00830 }
```

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 131 of file low_saurion.c.

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

```
    *r = temp;
  }
  struct request *req = *r;
  req->iovec_count = (int)amount;
  void **children_ptr = (void **)malloc(amount * sizeof(void *));
  if (!children_ptr) {
    free_request(req, children_ptr, 0);
    return ERROR_CODE;
  }
  for (size_t i = 0; i < amount; ++i) {
    if (!allocate_iovec(&req->iov[i], amount, i, full_size, children_ptr)) {
      free_request(req, children_ptr, amount);
      return ERROR_CODE;
    }
    if (!initialize_iovec(&req->iov[i], amount, i, m, s, h)) {
      free_request(req, children_ptr, amount);
      return ERROR_CODE;
    }
  }
  if (list_insert(l, req, amount, children_ptr)) {
    free_request(req, children_ptr, amount);
    return ERROR_CODE;
  }
  free(children_ptr);
  return SUCCESS_CODE;
}
```

Chapter 6

Class Documentation

6.1 ThreadPool::AsyncMultiQueue Struct Reference

Public Member Functions

- AsyncMultiQueue ()
- ∼AsyncMultiQueue ()
- AsyncMultiQueue (const AsyncMultiQueue &)=delete
- AsyncMultiQueue & operator= (const AsyncMultiQueue &)=delete
- AsyncMultiQueue (AsyncMultiQueue &&)=delete
- AsyncMultiQueue & operator= (AsyncMultiQueue &&)=delete
- void new_queue (uint32_t qid, uint32_t cnt)
- void remove_queue (uint32_t qid)
- void push (uint32_t qid, void(∗nfn)(void ∗), void ∗arg)
- Task ∗ front (uint32_t &qid)
- void pop (uint32_t qid)
- void clear ()
- bool empty ()

Private Attributes

- std::unordered_map< uint32_t, AsyncQueue ∗ > m_queues
- std::unordered_map< uint32_t, AsyncQueue ∗ >::iterator m_it

6.1.1 Detailed Description

Definition at line 50 of file threadpool.hpp.

6.1.2 Constructor & Destructor Documentation

6.1.2.1 AsyncMultiQueue() [1/3]

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

Definition at line 52 of file threadpool.cpp.

```
00052                                      {
00053   new_queue(0, 0);
00054   m_it = m_queues.begin();
00055 }
```

6.1.2.2 ∼AsyncMultiQueue()

```
TP::AsyncMultiQueue::∼AsyncMultiQueue ( )
```

Definition at line 56 of file threadpool.cpp.

```
00056                                       {
00057   for (auto& queue :  m_queues) {
00058     delete queue.second;
00059   }
00060 }
```

6.1.2.3 AsyncMultiQueue() [2/3]

```
ThreadPool::AsyncMultiQueue::AsyncMultiQueue (
              const AsyncMultiQueue &  )  [delete]
```

6.1.2.4 AsyncMultiQueue() [3/3]

```
ThreadPool::AsyncMultiQueue::AsyncMultiQueue (
              AsyncMultiQueue &&  )  [delete]
```

6.1.3 Member Function Documentation

6.1.3.1 clear()

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

Definition at line 101 of file threadpool.cpp.

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

6.1.3.2 empty()

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

Definition at line 115 of file threadpool.cpp.

```
00115                                {
00116   for (auto& queue :  m_queues) {
00117     if (!queue.second->empty()) {
00118       return false;
00119     }
00120   }
00121   return true;
00122 }
```

6.1.3.3 front()

```
Task * TP::AsyncMultiQueue::front (
              uint32_t & qid )
```

Definition at line 80 of file threadpool.cpp.

```
00080                                             {
00081   if (empty()) {
00082     throw std::out_of_range("empty queue");
00083   }
00084   auto newit = m_it;
00085   ++newit;
00086   while (newit != m_it) {
00087     if (newit == m_queues.end()) {
00088       newit = m_queues.begin();
00089     }
00090     if (!newit->second->empty()) {
00091       break;
00092     }
00093     ++newit;
00094   }
00095   Task* task = newit->second->front();
00096   qid = newit->first;
00097   m_it = newit;
00098   return task;
00099 }
```

6.1.3.4 new_queue()

```
void TP::AsyncMultiQueue::new_queue (
              uint32_t qid,
              uint32_t cnt )
```

Definition at line 62 of file threadpool.cpp.

```
00062                                                             {
00063   if (m_queues.find(qid) != m_queues.end()) {
00064     throw std::out_of_range("queue already exists");
00065   }
00066   m_queues.emplace(qid, new TP::AsyncQueue(cnt));
00067 }
```

6.1.3.5 operator=() [1/2]

```
AsyncMultiQueue & ThreadPool::AsyncMultiQueue::operator= (
              AsyncMultiQueue &&  )  [delete]
```

6.1.3.6 operator=() [2/2]

```
AsyncMultiQueue & ThreadPool::AsyncMultiQueue::operator= (
             const AsyncMultiQueue & )  [delete]
```

6.1.3.7 pop()

```
void TP::AsyncMultiQueue::pop (
             uint32_t qid )
```

Definition at line 100 of file threadpool.cpp.

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

6.1.3.8 push()

```
void TP::AsyncMultiQueue::push (
             uint32_t qid,
             void(*)(void *) nfn,
             void * arg )
```

Definition at line 77 of file threadpool.cpp.

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

6.1.3.9 remove_queue()

```
void TP::AsyncMultiQueue::remove_queue (
             uint32_t qid )
```

Definition at line 68 of file threadpool.cpp.

```
00068                                               {
00069   auto queue = m_queues.find(qid);
00070   if (queue == m_queues.end()) {
00071     throw std::out_of_range("queue not found");
00072   }
00073   delete queue->second;
00074   m_queues.erase(qid);
00075 }
```

6.1.4 Member Data Documentation

6.1.4.1 m_it

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

Definition at line 53 of file threadpool.hpp.

6.1.4.2 m_queues

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

Definition at line 52 of file threadpool.hpp.

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

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

6.2 ThreadPool::AsyncQueue Struct Reference

Public Member Functions

- AsyncQueue (uint32_t cnt)
- ∼AsyncQueue ()
- AsyncQueue (const AsyncQueue &)=delete
- AsyncQueue & operator= (const AsyncQueue &)=delete
- AsyncQueue (AsyncQueue &&)=delete
- AsyncQueue & operator= (AsyncQueue &&)=delete
- void push (Task ∗task)
- Task ∗ front ()
- void pop ()
- bool empty ()

Private Attributes

- std::queue< Task ∗ > m_queue
- uint32_t m_max
- uint32_t m_cnt
- pthread_mutex_t m_mtx = PTHREAD_MUTEX_INITIALIZER

6.2.1 Detailed Description

Definition at line 29 of file threadpool.hpp.

6.2.2 Constructor & Destructor Documentation

6.2.2.1 AsyncQueue() [1/3]

```
TP::AsyncQueue::AsyncQueue (
            uint32_t cnt )  [explicit]
```

Definition at line 16 of file threadpool.cpp.

```
00016 :  m_max(cnt), m_cnt(0) {}
```

6.2.2.2 ∼AsyncQueue()

```
TP::AsyncQueue::∼AsyncQueue ( )
```

Definition at line 18 of file threadpool.cpp.

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

6.2.2.3 AsyncQueue() [2/3]

```
ThreadPool::AsyncQueue::AsyncQueue (
            const AsyncQueue &  )  [delete]
```

6.2.2.4 AsyncQueue() [3/3]

```
ThreadPool::AsyncQueue::AsyncQueue (
            AsyncQueue &&  )  [delete]
```

6.2.3 Member Function Documentation

6.2.3.1 empty()

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

Definition at line 43 of file threadpool.cpp.

```
00043                          {
00044   pthread_mutex_lock(&m_mtx);
00045   bool empty = m_queue.empty();
00046   pthread_mutex_unlock(&m_mtx);
00047   return empty;
00048 }
```

6.2.3.2 front()

```
Task * TP::AsyncQueue::front ( )
```

Definition at line 25 of file threadpool.cpp.

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

6.2.3.3 operator=() [1/2]

```
AsyncQueue & ThreadPool::AsyncQueue::operator= (
               AsyncQueue &&   )   [delete]
```

6.2.3.4 operator=() [2/2]

```
AsyncQueue & ThreadPool::AsyncQueue::operator= (
               const AsyncQueue &   )   [delete]
```

6.2.3.5 pop()

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

Definition at line 37 of file threadpool.cpp.

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

6.2.3.6 push()

```
void TP::AsyncQueue::push (
               Task * task )
```

Definition at line 20 of file threadpool.cpp.

```
00020                                         {
00021   pthread_mutex_lock(&m_mtx);
00022   m_queue.push(task);
00023   pthread_mutex_unlock(&m_mtx);
00024 }
```

6.2.4 Member Data Documentation

6.2.4.1 m_cnt

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

Definition at line 33 of file threadpool.hpp.

6.2.4.2 m_max

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

Definition at line 32 of file threadpool.hpp.

6.2.4.3 m_mtx

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

Definition at line 34 of file threadpool.hpp.

6.2.4.4 m_queue

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

Definition at line 31 of file threadpool.hpp.

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

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

6.3 Node Struct Reference

Collaboration diagram for Node:

Public Attributes

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

6.3.1 Detailed Description

Definition at line 6 of file linked_list.c.

6.3.2 Member Data Documentation

6.3.2.1 children

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

Definition at line 9 of file linked_list.c.

6.3.2.2 next

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

Definition at line 10 of file linked_list.c.

6.3.2.3 ptr

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

Definition at line 7 of file linked_list.c.

6.3.2.4 size

```
size_t Node::size
```

Definition at line 8 of file linked_list.c.

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

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

6.4 request Struct Reference

Public Attributes

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

6.4.1 Detailed Description

Definition at line 15 of file low_saurion.c.

6.4.2 Member Data Documentation

6.4.2.1 client_socket

```
int request::client_socket
```

Definition at line 23 of file low_saurion.c.

6.4.2.2 event_type

```
int request::event_type
```

Definition at line 21 of file low_saurion.c.

6.4.2.3 iov

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

Definition at line 24 of file low_saurion.c.

6.4.2.4 iovec_count

```
size_t request::iovec_count
```

Definition at line 22 of file low_saurion.c.

6.4.2.5 next_iov

```
size_t request::next_iov
```

Definition at line 19 of file low_saurion.c.

6.4.2.6 next_offset

```
size_t request::next_offset
```

Definition at line 20 of file low_saurion.c.

6.4.2.7 prev

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

Definition at line 16 of file low_saurion.c.

6.4.2.8 prev_remain

```
size_t request::prev_remain
```

Definition at line 18 of file low_saurion.c.

6.4.2.9 prev_size

```
size_t request::prev_size
```

Definition at line 17 of file low_saurion.c.

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

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

6.5 saurion Struct Reference

Main structure for managing io_uring and socket events.

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

Collaboration diagram for saurion:

Classes

- struct saurion_callbacks

  Structure containing callback functions to handle socket events.

Public Attributes

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

6.5.1 Detailed Description

Main structure for managing io_uring and socket events.

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

Definition at line 137 of file low_saurion.h.

6.5.2 Member Data Documentation

6.5.2.1 efds

```
int∗ saurion::efds
```

Eventfd descriptors used for internal signaling between threads.

Definition at line 142 of file low_saurion.h.

6.5.2.2 list

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

Linked list for storing active requests.

Definition at line 143 of file low_saurion.h.

6.5.2.3 m_rings

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

Array of mutexes to protect the io_uring rings during concurrent access.

Definition at line 140 of file low_saurion.h.

6.5.2.4 n_threads

```
uint32_t saurion::n_threads
```

Number of threads in the thread pool.

Definition at line 148 of file low_saurion.h.

6.5.2.5 next

```
uint32_t saurion::next
```

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

Definition at line 149 of file low_saurion.h.

6.5.2.6 pool

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

Thread pool for executing tasks in parallel.

Definition at line 147 of file low_saurion.h.

6.5.2.7 rings

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

Array of io_uring structures for managing the event queue.

Definition at line 138 of file low_saurion.h.

6.5.2.8 ss

```
int saurion::ss
```

Server socket descriptor for accepting connections.

Definition at line 141 of file low_saurion.h.

6.5.2.9 status

```
int saurion::status
```

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

Definition at line 146 of file low_saurion.h.

6.5.2.10 status_c

```
pthread_cond_t saurion::status_c
```

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

Definition at line 145 of file low_saurion.h.

6.5.2.11 status_m

```
pthread_mutex_t saurion::status_m
```

Mutex to protect the state of the structure.

Definition at line 144 of file low_saurion.h.

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

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

6.6 Saurion Class Reference

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

Collaboration diagram for Saurion:

Public Types

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

Public Member Functions

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

Private Attributes

- struct saurion ∗ s

6.6.1 Detailed Description

Definition at line 6 of file saurion.hpp.

6.6.2 Member Typedef Documentation

6.6.2.1 ClosedCb

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

Definition at line 11 of file saurion.hpp.

6.6.2.2 ConnectedCb

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

Definition at line 8 of file saurion.hpp.

6.6.2.3 ErrorCb

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

Definition at line 12 of file saurion.hpp.

6.6.2.4 ReadedCb

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

Definition at line 9 of file saurion.hpp.

6.6.2.5 WroteCb

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

Definition at line 10 of file saurion.hpp.

6.6.3 Constructor & Destructor Documentation

6.6.3.1 Saurion() [1/3]

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

Definition at line 7 of file saurion.cpp.

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

6.6.3.2 ∼Saurion()

```
Saurion::∼Saurion ( )
```

Definition at line 15 of file saurion.cpp.

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

6.6.3.3 Saurion() [2/3]

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

6.6.3.4 Saurion() [3/3]

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

6.6.4 Member Function Documentation

6.6.4.1 init()

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

Definition at line 17 of file saurion.cpp.

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

6.6.4.2 on_closed()

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

Definition at line 43 of file saurion.cpp.

```
00043                                                                                {
00044   s->cb.on_closed = ncb;
00045   s->cb.on_closed_arg = arg;
00046   return this;
00047 }
```

6.6.4.3 on_connected()

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

Definition at line 25 of file saurion.cpp.

```
00025 {
00026   s->cb.on_connected = ncb;
00027   s->cb.on_connected_arg = arg;
00028   return this;
00029 }
```

6.6.4.4 on_error()

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

Definition at line 49 of file saurion.cpp.

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

6.6.4.5 on_readed()

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

Definition at line 31 of file saurion.cpp.

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

6.6.4.6 on_wrote()

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

Definition at line 37 of file saurion.cpp.

```
00037 {
00038   s->cb.on_wrote = ncb;
00039   s->cb.on_wrote_arg = arg;
00040   return this;
00041 }
```

6.6.4.7 operator=() [1/2]

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

6.6.4.8 operator=() [2/2]

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

6.6.4.9 send()

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

Definition at line 55 of file saurion.cpp.

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

6.6.4.10 stop()

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

Definition at line 23 of file saurion.cpp.

```
00023 { saurion_stop(this->s); }
```

6.6.5 Member Data Documentation

6.6.5.1 s

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

Definition at line 34 of file saurion.hpp.

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

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

6.7 saurion::saurion_callbacks Struct Reference

Structure containing callback functions to handle socket events.

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

Public Attributes

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

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

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

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

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

    Callback for handling error events.
- void ∗ on_error_arg

6.7.1 Detailed Description

Structure containing callback functions to handle socket events.

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

Definition at line 159 of file low_saurion.h.

6.7.2 Member Data Documentation

6.7.2.1 on_closed

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

Callback for handling socket closures.

Parameters

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

Definition at line 195 of file low_saurion.h.

6.7.2.2 on_closed_arg

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

Additional argument for the close callback.

Definition at line 196 of file low_saurion.h.

6.7.2.3 on_connected

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

Callback for handling new connections.

Parameters

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

Definition at line 166 of file low_saurion.h.

6.7.2.4 on_connected_arg

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

Additional argument for the connection callback.

Definition at line 167 of file low_saurion.h.

6.7.2.5 on_error

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

Callback for handling error events.

Parameters

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

Definition at line 206 of file low_saurion.h.

6.7.2.6 on_error_arg

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

Additional argument for the error callback.

Definition at line 207 of file low_saurion.h.

6.7.2.7 on_readed

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

Callback for handling read events.

Parameters

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

Definition at line 177 of file low_saurion.h.

6.7.2.8 on_readed_arg

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

Additional argument for the read callback.

Definition at line 178 of file low_saurion.h.

6.7.2.9 on_wrote

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

Callback for handling write events.

Parameters

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

Definition at line 186 of file low_saurion.h.

6.7.2.10 on_wrote_arg

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

Additional argument for the write callback.

Definition at line 187 of file low_saurion.h.

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

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

6.8 saurion_callbacks Struct Reference

Structure containing callback functions to handle socket events.

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

Public Attributes

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

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

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

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

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

  Callback for handling error events.
- void ∗ on_error_arg

6.8.1 Detailed Description

Structure containing callback functions to handle socket events.

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

Definition at line 21 of file low_saurion.h.

6.8.2 Member Data Documentation

6.8.2.1 on_closed

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

Callback for handling socket closures.

Parameters

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

Definition at line 57 of file low_saurion.h.

6.8.2.2 on_closed_arg

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

Additional argument for the close callback.

Definition at line 58 of file low_saurion.h.

6.8.2.3 on_connected

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

Callback for handling new connections.

Parameters

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

Definition at line 28 of file low_saurion.h.

6.8.2.4 on_connected_arg

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

Additional argument for the connection callback.

Definition at line 29 of file low_saurion.h.

6.8.2.5 on_error

```
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 68 of file low_saurion.h.

6.8.2.6 on_error_arg

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

Additional argument for the error callback.

Definition at line 69 of file low_saurion.h.

6.8.2.7 on_readed

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

Callback for handling read events.

Parameters

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

Definition at line 39 of file low_saurion.h.

6.8.2.8 on_readed_arg

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

Additional argument for the read callback.

Definition at line 40 of file low_saurion.h.

6.8.2.9 on_wrote

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

Callback for handling write events.

Parameters

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

Definition at line 48 of file low_saurion.h.

6.8.2.10 on_wrote_arg

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

Additional argument for the write callback.

Definition at line 49 of file low_saurion.h.

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

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

6.9 saurion_wrapper Struct Reference

Collaboration diagram for saurion_wrapper:

Public Attributes

- struct saurion ∗ s
- uint32_t sel

6.9.1 Detailed Description

Definition at line 31 of file low_saurion.c.

6.9.2 Member Data Documentation

6.9.2.1 s

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

Definition at line 32 of file low_saurion.c.

6.9.2.2 sel

```
uint32_t saurion_wrapper::sel
```

Definition at line 33 of file low_saurion.c.

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

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

6.10 Task Struct Reference

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

Public Member Functions

- Task (void(∗nfn)(void ∗), void ∗narg)
- Task (const Task &)=delete
- Task (Task &&)=delete
- Task & operator= (const Task &)=delete
- Task & operator= (Task &&)=delete
- ∼Task ()=default

Public Attributes

- void(∗ function)(void ∗)
- void ∗ argument

6.10.1 Detailed Description

Definition at line 14 of file threadpool.hpp.

6.10.2 Constructor & Destructor Documentation

6.10.2.1 Task() [1/3]

```
T::Task (
             void(*)(void *) nfn,
             void * narg )   [explicit]
```

Definition at line 13 of file threadpool.cpp.

```
00013 :  function(nfn), argument(narg) {}
```

6.10.2.2 Task() [2/3]

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

6.10.2.3 Task() [3/3]

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

6.10.2.4 ∼Task()

```
Task::∼Task ( )  [default]
```

6.10.3 Member Function Documentation

6.10.3.1 operator=() [1/2]

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

6.10.3.2 operator=() [2/2]

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

6.10.4 Member Data Documentation

6.10.4.1 argument

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

Definition at line 16 of file threadpool.hpp.

6.10.4.2 function

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

Definition at line 15 of file threadpool.hpp.

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

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

6.11 ThreadPool Class Reference

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

Collaboration diagram for ThreadPool:

Classes

- struct AsyncMultiQueue
- struct AsyncQueue

Public Member Functions

- ThreadPool ()
- ThreadPool (size_t num_threads)
- ThreadPool (const ThreadPool &)=delete
- ThreadPool & operator= (const ThreadPool &)=delete
- ThreadPool (ThreadPool &&)=delete
- ThreadPool & operator= (ThreadPool &&)=delete
- void init ()
- void stop ()
- void add (uint32_t qid, void(∗nfn)(void ∗), void ∗arg)
- void add (void(∗nfn)(void ∗), void ∗arg)
- void new_queue (uint32_t qid, uint32_t cnt)
- void remove_queue (uint32_t qid)
- bool empty ()
- void wait_empty ()
- ∼ThreadPool ()

Private Types

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

Private Member Functions

- void wait_closeable ()
- void thread_worker ()

Static Private Member Functions

- static void ∗ thread_entry (void ∗arg)

Private Attributes

- size_t m_nth
- size_t m_started
- AsyncMultiQueue m_queues
- pthread_mutex_t m_q_mtx = PTHREAD_MUTEX_INITIALIZER
- pthread_cond_t m_q_cond = PTHREAD_COND_INITIALIZER
- pthread_t ∗ m_ths
- volatile sig_atomic_t m_fstop
- volatile sig_atomic_t m_faccept

Static Private Attributes

- static pthread_mutex_t s_mtx = PTHREAD_MUTEX_INITIALIZER

6.11.1 Detailed Description

Definition at line 27 of file threadpool.hpp.

6.11.2 Member Typedef Documentation

6.11.2.1 AsyncMultiQueue

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

6.11.2.2 AsyncQueue

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

6.11.3 Constructor & Destructor Documentation

6.11.3.1 ThreadPool() [1/4]

```
TP::ThreadPool ( )
```

Definition at line 128 of file threadpool.cpp.

```
00128 :   ThreadPool(4) {}
```

6.11.3.2 ThreadPool() [2/4]

```
TP::ThreadPool (
              size_t num_threads )  [explicit]
```

Definition at line 130 of file threadpool.cpp.

```
00131     :  m_nth(num_threads < 2 ?  2 :  num_threads),
00132       m_started(0),
00133       m_ths(new pthread_t[m_nth]{0}),
00134       m_fstop(0),
00135       m_faccept(0) {}
```

6.11.3.3 ThreadPool() [3/4]

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

6.11.3.4 ThreadPool() [4/4]

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

6.11.3.5 ∼ThreadPool()

```
TP::∼ThreadPool ( )
```

Definition at line 233 of file threadpool.cpp.

```
00233                  {
00234   stop();
00235   m_queues.clear();
00236   delete[] m_ths;
00237   pthread_mutex_destroy(&s_mtx);
00238 }
```

6.11.4 Member Function Documentation

6.11.4.1 add() [1/2]

```
void TP::add (
            uint32_t qid,
            void(*)(void *) nfn,
            void * arg )
```

Definition at line 169 of file threadpool.cpp.

```
00169                                                          {
00170   if (m_faccept == 0) {
00171     throw std::logic_error("threadpool already closed");
00172   }
00173   if (nfn == nullptr) {
00174     throw std::logic_error("function pointer cannot be null");
00175   }
00176   bool failed = false;
00177   pthread_mutex_lock(&m_q_mtx);
00178   try {
00179     m_queues.push(qid, nfn, arg);
00180   } catch (const std::out_of_range& e) {
00181     failed = true;
00182   }
00183   pthread_cond_signal(&m_q_cond);
00184   pthread_mutex_unlock(&m_q_mtx);
00185   if (failed) {
00186     throw std::out_of_range("queue not found");
00187   }
00188 }
```

6.11.4.2 add() [2/2]

```
void TP::add (
            void(*)(void *) nfn,
            void * arg )
```

Definition at line 189 of file threadpool.cpp.

```
00189                                     {
00190   add(0, nfn, arg);  // Agregar a la cola por defecto
00191 }
```

6.11.4.3 empty()

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

Definition at line 218 of file threadpool.cpp.

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

6.11.4.4 init()

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

Definition at line 137 of file threadpool.cpp.

```
00137             {
00138   if (m_started != 0) {
00139     return;
00140   }
00141   m_faccept = 1;
00142   m_fstop = 0;
00143   pthread_mutex_lock(&s_mtx);
00144   for (size_t i = 0; i < m_nth; ++i) {
00145     pthread_create(&m_ths[i], nullptr, thread_entry, this);
00146     ++m_started;
00147   }
00148   pthread_mutex_unlock(&s_mtx);
00149 }
```

6.11.4.5 new_queue()

```
void TP::new_queue (
            uint32_t qid,
            uint32_t cnt )
```

Definition at line 192 of file threadpool.cpp.

```
00192                                        {
00193   pthread_mutex_lock(&m_q_mtx);
00194   try {
00195     m_queues.new_queue(qid, cnt);
00196   } catch (const std::out_of_range& e) {
00197     pthread_mutex_unlock(&m_q_mtx);
00198     throw e;
00199   }
00200   pthread_mutex_unlock(&m_q_mtx);
00201 }
```

6.11.4.6 operator=() [1/2]

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

6.11.4.7 operator=() [2/2]

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

6.11.4.8 remove_queue()

```
void TP::remove_queue (
              uint32_t qid )
```

Definition at line 202 of file threadpool.cpp.

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

6.11.4.9 stop()

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

Definition at line 150 of file threadpool.cpp.

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

6.11.4.10 thread_entry()

```
void * TP::thread_entry (
             void * arg )  [static], [private]
```

Definition at line 265 of file threadpool.cpp.

```
00265                                   {
00266   auto* pool = static_cast<TP*>(arg);
00267   pool->thread_worker();
00268   return nullptr;
00269 }
```

6.11.4.11 thread_worker()

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

Definition at line 240 of file threadpool.cpp.

```
00240                          {
00241   // Lógica del trabajador del hilo
00242   uint32_t qid = 0;
00243   while (m_fstop == 0) {
00244     // Buscar una tarea para ejecutar
00245     try {
00246       pthread_mutex_lock(&m_q_mtx);
00247       Task* task = m_queues.front(qid);
00248       pthread_cond_signal(&m_q_cond);
00249       pthread_mutex_unlock(&m_q_mtx);
00250       try {
00251         task->function(task->argument);
00252       } catch (...)  {
00253       }
00254       delete task;  // TODO delete task
00255       pthread_mutex_lock(&m_q_mtx);
00256       m_queues.pop(qid);
00257       pthread_cond_broadcast(&m_q_cond);
00258       pthread_mutex_unlock(&m_q_mtx);
00259     } catch (const std::out_of_range& e) {
00260       pthread_mutex_unlock(&m_q_mtx);
00261       wait_closeable();
00262     }
00263   }
00264 }
```

6.11.4.12 wait_closeable()

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

Definition at line 219 of file threadpool.cpp.

```
00219                           {
00220   pthread_mutex_lock(&m_q_mtx);
00221   while (empty() && m_fstop == 0) {
00222     pthread_cond_wait(&m_q_cond, &m_q_mtx);
00223   }
00224   pthread_mutex_unlock(&m_q_mtx);
00225 }
```

6.11.4.13 wait_empty()

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

Definition at line 226 of file threadpool.cpp.

```
00226                          {
00227   pthread_mutex_lock(&m_q_mtx);
00228   while (!m_queues.empty()) {
00229     pthread_cond_wait(&m_q_cond, &m_q_mtx);
00230   }
00231   pthread_mutex_unlock(&m_q_mtx);
00232 }
```

6.11.5 Member Data Documentation

6.11.5.1 m_faccept

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

Definition at line 106 of file threadpool.hpp.

6.11.5.2 m_fstop

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

Definition at line 105 of file threadpool.hpp.

6.11.5.3 m_nth

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

Definition at line 98 of file threadpool.hpp.

6.11.5.4 m_q_cond

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

Definition at line 102 of file threadpool.hpp.

6.11.5.5 m_q_mtx

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

Definition at line 101 of file threadpool.hpp.

6.11.5.6 m_queues

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

Definition at line 100 of file threadpool.hpp.

6.11.5.7 m_started

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

Definition at line 99 of file threadpool.hpp.

6.11.5.8 m_ths

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

Definition at line 104 of file threadpool.hpp.

6.11.5.9 s_mtx

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

Definition at line 103 of file threadpool.hpp.

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

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

Chapter 7

File Documentation

7.1 /__w/saurion/saurion/include/cthreadpool.hpp File Reference

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

Typedefs

- typedef struct ThreadPool ThreadPool

Functions

- ThreadPool ∗ ThreadPool_create (size_t num_threads)
- ThreadPool ∗ ThreadPool_create_default (void)
- void ThreadPool_init (ThreadPool ∗thp)
- void ThreadPool_stop (ThreadPool ∗thp)
- void ThreadPool_add (ThreadPool ∗thp, uint32_t qid, void(∗nfn)(void ∗), void ∗arg)
- void ThreadPool_add_default (ThreadPool ∗thp, void(∗nfn)(void ∗), void ∗arg)
- void ThreadPool_new_queue (ThreadPool ∗thp, uint32_t qid, uint32_t cnt)
- void ThreadPool_remove_queue (ThreadPool ∗thp, uint32_t qid)
- bool ThreadPool_empty (ThreadPool ∗thp)
- void ThreadPool_wait_empty (ThreadPool ∗thp)
- void ThreadPool_destroy (ThreadPool ∗thp)

7.1.1 Typedef Documentation

7.1.1.1 ThreadPool

```
typedef struct ThreadPool ThreadPool
```

Definition at line 8 of file cthreadpool.hpp.

7.1.2 Function Documentation

7.1.2.1 ThreadPool_add()

```
void ThreadPool_add (
            ThreadPool * thp,
            uint32_t qid,
            void(*)(void *) nfn,
            void * arg )
```

Definition at line 11 of file cthreadpool.cpp.

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

7.1.2.2 ThreadPool_add_default()

```
void ThreadPool_add_default (
            ThreadPool * thp,
            void(*)(void *) nfn,
            void * arg )
```

Definition at line 13 of file cthreadpool.cpp.

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

7.1.2.3 ThreadPool_create()

```
ThreadPool * ThreadPool_create (
            size_t num_threads )
```

Definition at line 3 of file cthreadpool.cpp.

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

7.1.2.4 ThreadPool_create_default()

```
ThreadPool * ThreadPool_create_default (
            void )
```

Definition at line 5 of file cthreadpool.cpp.

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

7.1.2.5 ThreadPool_destroy()

```
void ThreadPool_destroy (
            ThreadPool * thp )
```

Definition at line 23 of file cthreadpool.cpp.

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

7.1.2.6 ThreadPool_empty()

```
bool ThreadPool_empty (
            ThreadPool * thp )
```

Definition at line 19 of file cthreadpool.cpp.

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

7.1.2.7 ThreadPool_init()

```
void ThreadPool_init (
            ThreadPool * thp )
```

Definition at line 7 of file cthreadpool.cpp.

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

7.1.2.8 ThreadPool_new_queue()

```
void ThreadPool_new_queue (
            ThreadPool * thp,
            uint32_t qid,
            uint32_t cnt )
```

Definition at line 15 of file cthreadpool.cpp.

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

7.1.2.9 ThreadPool_remove_queue()

```
void ThreadPool_remove_queue (
            ThreadPool * thp,
            uint32_t qid )
```

Definition at line 17 of file cthreadpool.cpp.

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

7.1.2.10 ThreadPool_stop()

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

Definition at line 9 of file cthreadpool.cpp.

```
00009 { thp->stop(); }
```

7.1.2.11 ThreadPool_wait_empty()

```
void ThreadPool_wait_empty (
              ThreadPool * thp )
```

Definition at line 21 of file cthreadpool.cpp.

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

7.2 cthreadpool.hpp

Go to the documentation of this file.

```
#ifndef THREADPOOL_H
#define THREADPOOL_H

#ifdef __cplusplus
#include "threadpool.hpp"
extern "C" {
#else
typedef struct ThreadPool ThreadPool;
#endif  // __cplusplus

ThreadPool *ThreadPool_create(size_t num_threads);

ThreadPool *ThreadPool_create_default(void);

void ThreadPool_init(ThreadPool *thp);

void ThreadPool_stop(ThreadPool *thp);

void ThreadPool_add(ThreadPool *thp, uint32_t qid, void (*nfn)(void *), void *arg);

void ThreadPool_add_default(ThreadPool *thp, void (*nfn)(void *), void *arg);

void ThreadPool_new_queue(ThreadPool *thp, uint32_t qid, uint32_t cnt);

void ThreadPool_remove_queue(ThreadPool *thp, uint32_t qid);

bool ThreadPool_empty(ThreadPool *thp);

void ThreadPool_wait_empty(ThreadPool *thp);

void ThreadPool_destroy(ThreadPool *thp);

#ifdef __cplusplus
}
#endif  // __cplusplus

#endif  // !THREADPOOL_H
```

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

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

Include dependency graph for linked_list.h:

7.4 linked_list.h

Go to the documentation of this file.

```
#ifndef LINKED_LIST_H
#define LINKED_LIST_H

#ifdef __cplusplus
extern "C" {
#endif

#include <stddef.h>

struct Node;

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

void list_delete_node(struct Node** head, void* ptr);

void list_free(struct Node** head);

#ifdef __cplusplus
}
#endif

#endif  // !LINKED_LIST_H
```

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

```
#include <liburing.h>
#include <pthread.h>
#include "config.h"
#include "cthreadpool.hpp"
#include "linked_list.h"
```

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

Classes

- struct saurion

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

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

  Structure containing callback functions to handle socket events.

Macros

- #define PACKING_SZ 128

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

Functions

- int EXTERNAL_set_socket (int p)
- struct saurion ∗ saurion_create (uint32_t n_threads)

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

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

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

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

  Sends a message through a socket using io_uring.

Variables

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

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

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

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

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

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

7.5.1 Variable Documentation

7.5.1.1 efds

```
int∗ efds
```

Eventfd descriptors used for internal signaling between threads.

Definition at line 4 of file low_saurion.h.

7.5.1.2 list

```
struct Node∗ list
```

Linked list for storing active requests.

Definition at line 5 of file low_saurion.h.

7.5.1.3 m_rings

```
pthread_mutex_t* m_rings
```

Array of mutexes to protect the io_uring rings during concurrent access.

Definition at line 2 of file low_saurion.h.

7.5.1.4 n_threads

```
uint32_t n_threads
```

Number of threads in the thread pool.

Definition at line 10 of file low_saurion.h.

7.5.1.5 next

```
uint32_t next
```

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

Definition at line 11 of file low_saurion.h.

7.5.1.6 on_closed

```
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 35 of file low_saurion.h.

7.5.1.7 on_closed_arg

```
void * on_closed_arg
```

Additional argument for the close callback.

Definition at line 36 of file low_saurion.h.

7.5.1.8 on_connected

```
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.

7.5.1.9 on_connected_arg

```
void * on_connected_arg
```

Additional argument for the connection callback.

Definition at line 7 of file low_saurion.h.

7.5.1.10 on_error

```
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 46 of file low_saurion.h.

7.5.1.11 on_error_arg

```
void * on_error_arg
```

Additional argument for the error callback.

Definition at line 47 of file low_saurion.h.

7.5.1.12 on_readed

```
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 17 of file low_saurion.h.

7.5.1.13 on_readed_arg

```
void * on_readed_arg
```

Additional argument for the read callback.

Definition at line 18 of file low_saurion.h.

7.5.1.14 on_wrote

```
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 26 of file low_saurion.h.

7.5.1.15 on_wrote_arg

```
void * on_wrote_arg
```

Additional argument for the write callback.

Definition at line 27 of file low_saurion.h.

7.5.1.16 pool

```
ThreadPool* pool
```

Thread pool for executing tasks in parallel.

Definition at line 9 of file low_saurion.h.

7.5.1.17 rings

```
struct io_uring* rings
```

Array of io_uring structures for managing the event queue.

Definition at line 0 of file low_saurion.h.

7.5.1.18 ss

```
int ss
```

Server socket descriptor for accepting connections.

Definition at line 3 of file low_saurion.h.

7.5.1.19 status

```
int status
```

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

Definition at line 8 of file low_saurion.h.

7.5.1.20 status_c

```
pthread_cond_t status_c
```

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

Definition at line 7 of file low_saurion.h.

7.5.1.21 status_m

```
pthread_mutex_t status_m
```

Mutex to protect the state of the structure.

Definition at line 6 of file low_saurion.h.

7.6 low_saurion.h

Go to the documentation of this file.

```
00001
00098 #ifndef LOW_SAURION_H
00099 #define LOW_SAURION_H
00100
00101 #include <liburing.h>
00102 #include <pthread.h>
00103
00104 #include "config.h"
00105 #include "cthreadpool.hpp"
00106 #include "linked_list.h"
00107
00108 #ifdef __cplusplus
00109 extern "C" {
00110 #endif
00111
00129 #define PACKING_SZ 128
00130
00137 struct saurion {
00138   struct io_uring *rings;
00139   pthread_mutex_t
00140       *m_rings;
00141   int ss;
00142   int *efds;
00143   struct Node *list;
00144   pthread_mutex_t status_m;
00145   pthread_cond_t status_c;
00146   int status;
00147   ThreadPool *pool;
00148   uint32_t n_threads;
00149   uint32_t next;
00159   struct saurion_callbacks {
00166     void (*on_connected)(const int fd, void *arg);
```

```
    void *on_connected_arg;
    void (*on_readed)(const int fd, const void *const content, const ssize_t len, void *arg);
    void *on_readed_arg;
    void (*on_wrote)(const int fd, void *arg);
    void *on_wrote_arg;
    void (*on_closed)(const int fd, void *arg);
    void *on_closed_arg;
    void (*on_error)(const int fd, const char *const content, const ssize_t len, void *arg);
    void *on_error_arg;
  } __attribute__((aligned(PACKING_SZ))) cb;
} __attribute__((aligned(PACKING_SZ)));

int EXTERNAL_set_socket(int p);

[[nodiscard]]
struct saurion *saurion_create(uint32_t n_threads);

[[nodiscard]]
int saurion_start(struct saurion *s);

void saurion_stop(const struct saurion *s);

void saurion_destroy(struct saurion *s);

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

#ifdef __cplusplus
}
#endif

#endif  // !LOW_SAURION_H

```

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

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

Include dependency graph for low_saurion_secret.h:

Functions

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

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

7.8 low_saurion_secret.h

Go to the documentation of this file.

```
#ifndef LOW_SAURION_SECRET_H
#define LOW_SAURION_SECRET_H

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

#ifdef __cplusplus
extern "C" {
#endif
struct request {
  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[];
};
[[nodiscard]]
int allocate_iovec(struct iovec *iov, size_t amount, size_t pos, size_t size, void **chd_ptr);

[[nodiscard]]
int initialize_iovec(struct iovec *iov, size_t amount, size_t pos, const void *msg, size_t size,
                     uint8_t h);

[[nodiscard]]
int set_request(struct request **r, struct Node **l, size_t s, const void *m, uint8_t h);

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

void free_request(struct request *req, void **children_ptr, size_t amount);
#ifdef __cplusplus
}
#endif

#endif  // !LOW_SAURION_SECRET_H
```

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

`#include "low_saurion.h"`

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

Classes

- class Saurion

7.10 saurion.hpp

Go to the documentation of this file.

```
#ifndef SAURION_HPP
#define SAURION_HPP

#include "low_saurion.h"

class Saurion {
 public:
  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 *);

  explicit Saurion(const uint32_t thds, const int sck) noexcept;
  ~Saurion();

  Saurion(const Saurion &) = delete;
  Saurion(Saurion &&) = delete;
  Saurion &operator=(const Saurion &) = delete;
  Saurion &operator=(Saurion &&) = delete;

  void init() noexcept;
  void stop() noexcept;

  Saurion *on_connected(ConnectedCb ncb, void *arg) noexcept;
  Saurion *on_readed(ReadedCb ncb, void *arg) noexcept;
  Saurion *on_wrote(WroteCb ncb, void *arg) noexcept;
  Saurion *on_closed(ClosedCb ncb, void *arg) noexcept;
  Saurion *on_error(ErrorCb ncb, void *arg) noexcept;

  void send(const int fd, const char *const msg) noexcept;

 private:
  struct saurion *s;
};

#endif  // !SAURION_HPP
```

7.11 /__w/saurion/saurion/include/threadpool.hpp File Reference

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

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

Classes

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

Enumerations

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

7.11.1 Enumeration Type Documentation

7.11.1.1 StopFlags

```
enum StopFlags
```

Enumerator

KINDLY	
FORCE	

Definition at line 12 of file threadpool.hpp.

```
00012 { KINDLY = 0x01, FORCE = 0x02 };
```

7.12 threadpool.hpp

Go to the documentation of this file.

```
#ifndef THREADPOOL_HPP
#define THREADPOOL_HPP

// Estructura para representar una tarea
#include <bits/types/sig_atomic_t.h>
#include <pthread.h>

#include <cstdint>
#include <queue>
#include <unordered_map>

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

struct Task {
  void (*function)(void*);
  void* argument;

  explicit Task(void (*nfn)(void*), void* narg);
  Task(const Task&) = delete;
  Task(Task&&) = delete;
  Task& operator=(const Task&) = delete;
  Task& operator=(Task&&) = delete;
  ~Task() = default;
};

// Clase ThreadPool
class ThreadPool {
 private:
  typedef struct AsyncQueue {
   private:
    std::queue<Task*> m_queue;
    uint32_t m_max;
    uint32_t m_cnt;
    pthread_mutex_t m_mtx = PTHREAD_MUTEX_INITIALIZER;

   public:
    explicit AsyncQueue(uint32_t cnt);
    ~AsyncQueue();
    AsyncQueue(const AsyncQueue&) = delete;
    AsyncQueue& operator=(const AsyncQueue&) = delete;
    AsyncQueue(AsyncQueue&&) = delete;
    AsyncQueue& operator=(AsyncQueue&&) = delete;

    void push(Task* task);
    Task* front();
    void pop();

    bool empty();
  } AsyncQueue;
  typedef struct AsyncMultiQueue {
   private:
    std::unordered_map<uint32_t, AsyncQueue*> m_queues;
    std::unordered_map<uint32_t, AsyncQueue*>::iterator m_it;

   public:
    explicit AsyncMultiQueue();
    ~AsyncMultiQueue();
    AsyncMultiQueue(const AsyncMultiQueue&) = delete;
    AsyncMultiQueue& operator=(const AsyncMultiQueue&) = delete;
    AsyncMultiQueue(AsyncMultiQueue&&) = delete;
    AsyncMultiQueue& operator=(AsyncMultiQueue&&) = delete;

    void new_queue(uint32_t qid, uint32_t cnt);
    void remove_queue(uint32_t qid);

    void push(uint32_t qid, void (*nfn)(void*), void* arg);
    Task* front(uint32_t& qid);
    void pop(uint32_t qid);
    void clear();

    bool empty();
  } AsyncMultiQueue;

 public:
  ThreadPool();
  explicit ThreadPool(size_t num_threads);
  ThreadPool(const ThreadPool&) = delete;
  ThreadPool& operator=(const ThreadPool&) = delete;
  ThreadPool(ThreadPool&&) = delete;
  ThreadPool& operator=(ThreadPool&&) = delete;

  void init();
  void stop();
  void add(uint32_t qid, void (*nfn)(void*), void* arg);
  void add(void (*nfn)(void*), void* arg);
  void new_queue(uint32_t qid, uint32_t cnt);
  void remove_queue(uint32_t qid);
  bool empty();

 private:
  void wait_closeable();

 public:
  void wait_empty();
  ~ThreadPool();

 private:
  size_t m_nth;
  size_t m_started;
  AsyncMultiQueue m_queues;
  pthread_mutex_t m_q_mtx = PTHREAD_MUTEX_INITIALIZER;
  pthread_cond_t m_q_cond = PTHREAD_COND_INITIALIZER;
  static pthread_mutex_t s_mtx;
  pthread_t* m_ths;
  volatile sig_atomic_t m_fstop;
  volatile sig_atomic_t m_faccept;

  void thread_worker();
  static void* thread_entry(void* arg);
};

#endif
```

7.13 /__w/saurion/saurion/src/cthreadpool.cpp File Reference

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

Functions

- ThreadPool ∗ ThreadPool_create (size_t num_threads)
- ThreadPool ∗ ThreadPool_create_default (void)
- void ThreadPool_init (ThreadPool ∗thp)
- void ThreadPool_stop (ThreadPool ∗thp)
- void ThreadPool_add (ThreadPool ∗thp, uint32_t qid, void(∗nfn)(void ∗), void ∗arg)
- void ThreadPool_add_default (ThreadPool ∗thp, void(∗nfn)(void ∗), void ∗arg)
- void ThreadPool_new_queue (ThreadPool ∗thp, uint32_t qid, uint32_t cnt)
- void ThreadPool_remove_queue (ThreadPool ∗thp, uint32_t qid)
- bool ThreadPool_empty (ThreadPool ∗thp)
- void ThreadPool_wait_empty (ThreadPool ∗thp)
- void ThreadPool_destroy (ThreadPool ∗thp)

7.13.1 Function Documentation

7.13.1.1 ThreadPool_add()

```
void ThreadPool_add (
             ThreadPool * thp,
             uint32_t qid,
             void(*)(void *) nfn,
             void * arg )
```

Definition at line 11 of file cthreadpool.cpp.

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

7.13.1.2 ThreadPool_add_default()

```
void ThreadPool_add_default (
             ThreadPool * thp,
             void(*)(void *) nfn,
             void * arg )
```

Definition at line 13 of file cthreadpool.cpp.

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

7.13.1.3 ThreadPool_create()

```
ThreadPool * ThreadPool_create (
             size_t num_threads )
```

Definition at line 3 of file cthreadpool.cpp.

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

7.13.1.4 ThreadPool_create_default()

```
ThreadPool * ThreadPool_create_default (
             void )
```

Definition at line 5 of file cthreadpool.cpp.

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

7.13.1.5 ThreadPool_destroy()

```
void ThreadPool_destroy (
             ThreadPool * thp )
```

Definition at line 23 of file cthreadpool.cpp.

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

7.13.1.6 ThreadPool_empty()

```
bool ThreadPool_empty (
             ThreadPool * thp )
```

Definition at line 19 of file cthreadpool.cpp.

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

7.13.1.7 ThreadPool_init()

```
void ThreadPool_init (
            ThreadPool * thp )
```

Definition at line 7 of file cthreadpool.cpp.

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

7.13.1.8 ThreadPool_new_queue()

```
void ThreadPool_new_queue (
            ThreadPool * thp,
            uint32_t qid,
            uint32_t cnt )
```

Definition at line 15 of file cthreadpool.cpp.

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

7.13.1.9 ThreadPool_remove_queue()

```
void ThreadPool_remove_queue (
            ThreadPool * thp,
            uint32_t qid )
```

Definition at line 17 of file cthreadpool.cpp.

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

7.13.1.10 ThreadPool_stop()

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

Definition at line 9 of file cthreadpool.cpp.

```
00009 { thp->stop(); }
```

7.13.1.11 ThreadPool_wait_empty()

```
void ThreadPool_wait_empty (
            ThreadPool * thp )
```

Definition at line 21 of file cthreadpool.cpp.

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

7.14 cthreadpool.cpp

Go to the documentation of this file.

```
#include "cthreadpool.hpp"

ThreadPool *ThreadPool_create(size_t num_threads) { return new ThreadPool(num_threads); }

ThreadPool *ThreadPool_create_default(void) { return new ThreadPool(); }

void ThreadPool_init(ThreadPool *thp) { thp->init(); }

void ThreadPool_stop(ThreadPool *thp) { thp->stop(); }

void ThreadPool_add(ThreadPool *thp, uint32_t qid, void (*nfn)(void *), void *arg) { thp->add(qid, nfn, arg); }

void ThreadPool_add_default(ThreadPool *thp, void (*nfn)(void *), void *arg) { thp->add(nfn, arg); }

void ThreadPool_new_queue(ThreadPool *thp, uint32_t qid, uint32_t cnt) { thp->new_queue(qid, cnt); }

void ThreadPool_remove_queue(ThreadPool *thp, uint32_t qid) { thp->remove_queue(qid); }

bool ThreadPool_empty(ThreadPool *thp) { return thp->empty(); }

void ThreadPool_wait_empty(ThreadPool *thp) { thp->wait_empty(); }

void ThreadPool_destroy(ThreadPool *thp) { delete thp; }
```

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

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

Include dependency graph for linked_list.c:

Classes

- struct Node

Functions

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

Variables

- pthread_mutex_t list_mutex = PTHREAD_MUTEX_INITIALIZER

7.15.1 Function Documentation

7.15.1.1 create_node()

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

Definition at line 17 of file linked_list.c.

```
00017                                                                       {
00018   struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
00019   // LCOV_EXCL_START
00020   if (!new_node) {
00021     return NULL;
00022   }
00023   // LCOV_EXCL_STOP
00024   new_node->ptr = ptr;
00025   new_node->size = amount;
00026   new_node->children = NULL;
00027   if (amount > 0) {
00028     new_node->children = (struct Node **)malloc(sizeof(struct Node *) * amount);
00029     // LCOV_EXCL_START
00030     if (!new_node->children) {
00031       free(new_node);
00032       return NULL;
00033     }
00034     // LCOV_EXCL_STOP
00035     for (size_t i = 0; i < amount; ++i) {
00036       new_node->children[i] = (struct Node *)malloc(sizeof(struct Node));
00037 
00038       // LCOV_EXCL_START
00039       if (!new_node->children[i]) {
00040         for (size_t j = 0; j < i; ++j) {
00041           free(new_node->children[j]);
00042         }
00043         free(new_node);
00044         return NULL;
00045       }
00046     }
00047     // LCOV_EXCL_STOP
00048     for (size_t i = 0; i < amount; ++i) {
00049       new_node->children[i]->size = 0;
00050       new_node->children[i]->next = NULL;
00051       new_node->children[i]->ptr = children[i];
00052       new_node->children[i]->children = NULL;
00053     }
00054   }
00055   new_node->next = NULL;
00056   return new_node;
00057 }
```

7.15.1.2 free_node()

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

Definition at line 81 of file linked_list.c.

```
00081                                       {
00082   if (current->size > 0) {
00083     for (size_t i = 0; i < current->size; ++i) {
00084       free(current->children[i]->ptr);
00085       free(current->children[i]);
00086     }
00087     free(current->children);
00088   }
00089   free(current->ptr);
00090   free(current);
00091 }
```

7.15.1.3 list_delete_node()

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

Definition at line 93 of file linked_list.c.

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

7.15.1.4 list_free()

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

Definition at line 124 of file linked_list.c.

```
00124                                    {
00125   pthread_mutex_lock(&list_mutex);
00126   struct Node *current = *head;
00127   struct Node *next;
00128
00129   while (current) {
00130     next = current->next;
00131     free_node(current);
00132     current = next;
00133   }
00134
00135   *head = NULL;
00136   pthread_mutex_unlock(&list_mutex);
00137 }
```

7.15.1.5 list_insert()

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

Definition at line 59 of file linked_list.c.

```
00059                                                                                {
00060   struct Node *new_node = create_node(ptr, amount, children);
00061   // LCOV_EXCL_START
00062   if (!new_node) {
00063     return 1;
00064   }
00065   // LCOV_EXCL_STOP
00066   pthread_mutex_lock(&list_mutex);
00067   if (!*head) {
00068     *head = new_node;
00069     pthread_mutex_unlock(&list_mutex);
00070     return 0;
00071   }
00072   struct Node *temp = *head;
00073   while (temp->next) {
00074     temp = temp->next;
00075   }
00076   temp->next = new_node;
00077   pthread_mutex_unlock(&list_mutex);
00078   return 0;
00079 }
```

7.15.2 Variable Documentation

7.15.2.1 list_mutex

```
pthread_mutex_t list_mutex = PTHREAD_MUTEX_INITIALIZER
```

Definition at line 14 of file linked_list.c.

7.16 linked_list.c

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   void *ptr;
00008   size_t size;
00009   struct Node **children;
00010   struct Node *next;
00011 };
00012 
00013 // Global mutex for thread safety
00014 pthread_mutex_t list_mutex = PTHREAD_MUTEX_INITIALIZER;
00015 
00016 // Función para crear un nuevo nodo
00017 struct Node *create_node(void *ptr, size_t amount, void **children) {
00018   struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
00019   // LCOV_EXCL_START
00020   if (!new_node) {
00021     return NULL;
00022   }
00023   // LCOV_EXCL_STOP
00024   new_node->ptr = ptr;
00025   new_node->size = amount;
00026   new_node->children = NULL;
00027   if (amount > 0) {
00028     new_node->children = (struct Node **)malloc(sizeof(struct Node *) * amount);
00029     // LCOV_EXCL_START
00030     if (!new_node->children) {
      free(new_node);
      return NULL;
    }
    // LCOV_EXCL_STOP
    for (size_t i = 0; i < amount; ++i) {
      new_node->children[i] = (struct Node *)malloc(sizeof(struct Node));

      // LCOV_EXCL_START
      if (!new_node->children[i]) {
        for (size_t j = 0; j < i; ++j) {
          free(new_node->children[j]);
        }
        free(new_node);
        return NULL;
      }
    }
    // LCOV_EXCL_STOP
    for (size_t i = 0; i < amount; ++i) {
      new_node->children[i]->size = 0;
      new_node->children[i]->next = NULL;
      new_node->children[i]->ptr = children[i];
      new_node->children[i]->children = NULL;
    }
  }
  new_node->next = NULL;
  return new_node;
}

int list_insert(struct Node **head, void *ptr, size_t amount, void **children) {
  struct Node *new_node = create_node(ptr, amount, children);
  // LCOV_EXCL_START
  if (!new_node) {
    return 1;
  }
  // LCOV_EXCL_STOP
  pthread_mutex_lock(&list_mutex);
  if (!*head) {
    *head = new_node;
    pthread_mutex_unlock(&list_mutex);
    return 0;
  }
  struct Node *temp = *head;
  while (temp->next) {
    temp = temp->next;
  }
  temp->next = new_node;
  pthread_mutex_unlock(&list_mutex);
  return 0;
}

void free_node(struct Node *current) {
  if (current->size > 0) {
    for (size_t i = 0; i < current->size; ++i) {
      free(current->children[i]->ptr);
      free(current->children[i]);
    }
    free(current->children);
  }
  free(current->ptr);
  free(current);
}

void list_delete_node(struct Node **head, void *ptr) {
  pthread_mutex_lock(&list_mutex);
  struct Node *current = *head;
  struct Node *prev = NULL;

  // Si el nodo a eliminar es el nodo cabeza
  if (current && current->ptr == ptr) {
    *head = current->next;
    free_node(current);
    pthread_mutex_unlock(&list_mutex);
    return;
  }

  // Buscar el nodo a eliminar
  while (current && current->ptr != ptr) {
    prev = current;
    current = current->next;
  }

  // Si el dato no se encuentra en la lista
  if (!current) {
    pthread_mutex_unlock(&list_mutex);
    return;
  }

  // Desenlazar el nodo y liberarlo
  prev->next = current->next;
  free_node(current);
  pthread_mutex_unlock(&list_mutex);
}

void list_free(struct Node **head) {
  pthread_mutex_lock(&list_mutex);
  struct Node *current = *head;
  struct Node *next;

  while (current) {
    next = current->next;
    free_node(current);
    current = next;
  }

  *head = NULL;
  pthread_mutex_unlock(&list_mutex);
}
```

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

```
#include "low_saurion.h"
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/eventfd.h>
```

Include dependency graph for low_saurion.c:

Classes

- struct request
- struct saurion_wrapper

Macros

- #define EVENT_TYPE_ACCEPT 0
- #define EVENT_TYPE_READ 1
- #define EVENT_TYPE_WRITE 2
- #define EVENT_TYPE_WAIT 3
- #define EVENT_TYPE_ERROR 4
- #define MIN(a, b) ((a) < (b) ? (a) : (b))
- #define MAX(a, b) ((a) > (b) ? (a) : (b))

Functions

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

  Initializes a specified `iovec` structure with a message fragment.
- int allocate_iovec (struct iovec ∗iov, size_t amount, size_t pos, size_t size, void ∗∗chd_ptr)
- int set_request (struct request ∗∗r, struct Node ∗∗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, const struct sockaddr_in ∗const ca, socklen_t ∗const cal)
- static void add_efd (struct saurion ∗const s, const int client_socket, int sel)
- static void add_read (struct saurion ∗const s, const int client_socket)
- static void add_read_continue (struct saurion ∗const s, struct request ∗oreq, const int sel)
- static void add_write (struct saurion ∗const s, int fd, const char ∗const str, const int sel)
- static void handle_accept (const struct saurion ∗const s, const int fd)
- int read_chunk (void ∗∗dest, size_t ∗len, struct request ∗const req)

  Reads a message chunk from the request's iovec buffers, handling messages that may span multiple iovec entries.
- static void handle_read (struct saurion ∗const s, struct request ∗const 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 EXTERNAL_set_socket (const int p)
- struct saurion ∗ saurion_create (uint32_t n_threads)

  Creates an instance of the `saurion` structure.
- void saurion_worker_master (void ∗arg)
- void saurion_worker_slave (void ∗arg)
- int saurion_start (struct saurion ∗const s)

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

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

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

  Sends a message through a socket using io_uring.

Variables

- pthread_mutex_t print_mutex

7.17.1 Macro Definition Documentation

7.17.1.1 EVENT_TYPE_ACCEPT

```
#define EVENT_TYPE_ACCEPT 0
```

Definition at line 9 of file low_saurion.c.

7.17.1.2 EVENT_TYPE_ERROR

```
#define EVENT_TYPE_ERROR 4
```

Definition at line 13 of file low_saurion.c.

7.17.1.3 EVENT_TYPE_READ

```
#define EVENT_TYPE_READ 1
```

Definition at line 10 of file low_saurion.c.

7.17.1.4 EVENT_TYPE_WAIT

```
#define EVENT_TYPE_WAIT 3
```

Definition at line 12 of file low_saurion.c.

7.17.1.5 EVENT_TYPE_WRITE

```
#define EVENT_TYPE_WRITE 2
```

Definition at line 11 of file low_saurion.c.

7.17.1.6 MAX

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

Definition at line 28 of file low_saurion.c.

7.17.1.7 MIN

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

Definition at line 27 of file low_saurion.c.

7.17.2 Function Documentation

7.17.2.1 add_accept()

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

Definition at line 186 of file low_saurion.c.

```
00187                                              {
00188   int res = ERROR_CODE;
00189   pthread_mutex_lock(&s->m_rings[0]);
00190   while (res != SUCCESS_CODE) {
00191     struct io_uring_sqe *sqe = io_uring_get_sqe(&s->rings[0]);
00192     while (!sqe) {
00193       sqe = io_uring_get_sqe(&s->rings[0]);
00194       usleep(TIMEOUT_RETRY);
00195     }
00196     struct request *req = NULL;
00197     if (!set_request(&req, &s->list, 0, NULL, 0)) {
00198       free(sqe);
00199       usleep(TIMEOUT_RETRY);
00200       res = ERROR_CODE;
00201       continue;
00202     }
00203     req->client_socket = 0;
00204     req->event_type = EVENT_TYPE_ACCEPT;
00205     io_uring_prep_accept(sqe, s->ss, (struct sockaddr *)ca, cal, 0);
00206     io_uring_sqe_set_data(sqe, req);
00207     if (io_uring_submit(&s->rings[0]) < 0) {
00208       free(sqe);
00209       list_delete_node(&s->list, req);
00210       usleep(TIMEOUT_RETRY);
00211       res = ERROR_CODE;
00212       continue;
00213     }
00214     res = SUCCESS_CODE;
00215   }
00216   pthread_mutex_unlock(&s->m_rings[0]);
00217 }
```

7.17.2.2 add_efd()

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

Definition at line 219 of file low_saurion.c.

```
00219                                                                                  {
00220   pthread_mutex_lock(&s->m_rings[sel]);
00221   int res = ERROR_CODE;
00222   while (res != SUCCESS_CODE) {
00223     struct io_uring *ring = &s->rings[sel];
00224     struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
00225     while (!sqe) {
00226       sqe = io_uring_get_sqe(ring);
00227       usleep(TIMEOUT_RETRY);
00228     }
00229     struct request *req = NULL;
00230     if (!set_request(&req, &s->list, CHUNK_SZ, NULL, 0)) {
00231       free(sqe);
00232       res = ERROR_CODE;
00233       continue;
00234     }
00235     req->event_type = EVENT_TYPE_READ;
00236     req->client_socket = client_socket;
00237     io_uring_prep_readv(sqe, client_socket, &req->iov[0], req->iovec_count, 0);
00238     io_uring_sqe_set_data(sqe, req);
00239     if (io_uring_submit(ring) < 0) {
00240       free(sqe);
00241       list_delete_node(&s->list, req);
00242       res = ERROR_CODE;
00243       continue;
00244     }
00245     res = SUCCESS_CODE;
00246   }
00247   pthread_mutex_unlock(&s->m_rings[sel]);
00248 }
```

7.17.2.3 add_read()

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

Definition at line 250 of file low_saurion.c.

```
00250                                                                        {
00251   int sel = next(s);
00252   int res = ERROR_CODE;
00253   pthread_mutex_lock(&s->m_rings[sel]);
00254   while (res != SUCCESS_CODE) {
00255     struct io_uring *ring = &s->rings[sel];
00256     struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
00257     while (!sqe) {
00258       sqe = io_uring_get_sqe(ring);
00259       usleep(TIMEOUT_RETRY);
00260     }
00261     struct request *req = NULL;
00262     if (!set_request(&req, &s->list, CHUNK_SZ, NULL, 0)) {
00263       free(sqe);
00264       res = ERROR_CODE;
00265       continue;
00266     }
00267     req->event_type = EVENT_TYPE_READ;
00268     req->client_socket = client_socket;
00269     io_uring_prep_readv(sqe, client_socket, &req->iov[0], req->iovec_count, 0);
00270     io_uring_sqe_set_data(sqe, req);
00271     if (io_uring_submit(ring) < 0) {
00272       free(sqe);
00273       list_delete_node(&s->list, req);
00274       res = ERROR_CODE;
00275       continue;
00276     }
00277     res = SUCCESS_CODE;
00278   }
00279   pthread_mutex_unlock(&s->m_rings[sel]);
00280 }
```

7.17.2.4 add_read_continue()

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

Definition at line 282 of file low_saurion.c.

```
00282                                                                                                {
00283   pthread_mutex_lock(&s->m_rings[sel]);
00284   int res = ERROR_CODE;
00285   while (res != SUCCESS_CODE) {
00286     struct io_uring *ring = &s->rings[sel];
00287     struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
00288     while (!sqe) {
00289       sqe = io_uring_get_sqe(ring);
00290       usleep(TIMEOUT_RETRY);
00291     }
00292     if (!set_request(&oreq, &s->list, oreq->prev_remain, NULL, 0)) {
00293       free(sqe);
00294       res = ERROR_CODE;
00295       continue;
00296     }
00297     io_uring_prep_readv(sqe, oreq->client_socket, &oreq->iov[0], oreq->iovec_count, 0);
00298     io_uring_sqe_set_data(sqe, oreq);
00299     if (io_uring_submit(ring) < 0) {
00300       free(sqe);
00301       list_delete_node(&s->list, oreq);
00302       res = ERROR_CODE;
00303       continue;
00304     }
00305     res = SUCCESS_CODE;
00306   }
00307   pthread_mutex_unlock(&s->m_rings[sel]);
00308 }
```

7.17.2.5 add_write()

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

Definition at line 310 of file low_saurion.c.

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

7.17.2.6 handle_accept()

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

Definition at line 343 of file low_saurion.c.

```
00343                                                                    {
00344   if (s->cb.on_connected) {
00345     s->cb.on_connected(fd, s->cb.on_connected_arg);
00346   }
00347 }
```

7.17.2.7 handle_close()

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

Definition at line 540 of file low_saurion.c.

```
00540                                                                                            {
00541   if (s->cb.on_closed) {
00542     s->cb.on_closed(req->client_socket, s->cb.on_closed_arg);
00543   }
00544   close(req->client_socket);
00545 }
```

7.17.2.8 handle_error()

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

Definition at line 533 of file low_saurion.c.

```
00533                                                                                            {
00534   if (s->cb.on_error) {
00535     const char *resp = "ERROR";
00536     s->cb.on_error(req->client_socket, resp, (ssize_t)strlen(resp), s->cb.on_error_arg);
00537   }
00538 }
```

7.17.2.9 handle_read()

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

Definition at line 495 of file low_saurion.c.

```
00495                                                                               {
00496   void *msg = NULL;
00497   size_t len = 0;
00498   while (1) {
00499     if (!read_chunk(&msg, &len, req)) {
00500       break;
00501     }
00502     // Hay siguiente
00503     if (req->next_iov || req->next_offset) {
00504       if (s->cb.on_readed && msg) {
00505         s->cb.on_readed(req->client_socket, msg, len, s->cb.on_readed_arg);
00506       }
00507       free(msg);
00508       msg = NULL;
00509       continue;
00510     }
00511     // Hay previo pero no se ha completado
00512     if (req->prev && req->prev_size && req->prev_remain) {
00513       add_read_continue(s, req, next(s));
00514       return;
00515     }
00516     // Hay un único mensaje y se ha completado
00517     if (s->cb.on_readed && msg) {
00518       s->cb.on_readed(req->client_socket, msg, len, s->cb.on_readed_arg);
00519     }
00520     free(msg);
00521     msg = NULL;
00522     break;
00523   }
00524   add_read(s, req->client_socket);
00525 }
```

7.17.2.10 handle_write()

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

Definition at line 527 of file low_saurion.c.

```
00527                                                                    {
00528   if (s->cb.on_wrote) {
00529     s->cb.on_wrote(fd, s->cb.on_wrote_arg);
00530   }
00531 }
```

7.17.2.11 htonll()

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

Definition at line 41 of file low_saurion.c.

```
00041                                          {
00042   int num = 42;
00043   if (*(char *)&num == 42) {  // Little endian check
00044     uint32_t high_part = htonl((uint32_t)(value » 32));
00045     uint32_t low_part = htonl((uint32_t)(value & 0xFFFFFFFFLL));
00046     return ((uint64_t)low_part « 32) | high_part;
00047   }
00048   return value;
00049 }
```

7.17.2.12 next()

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

Definition at line 36 of file low_saurion.c.

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

7.17.2.13 ntohll()

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

Definition at line 51 of file low_saurion.c.

```
00051                                          {
00052   int num = 42;
00053   if (*(char *)&num == 42) {  // Little endian check
00054     uint32_t high_part = ntohl((uint32_t)(value » 32));
00055     uint32_t low_part = ntohl((uint32_t)(value & 0xFFFFFFFFLL));
00056     return ((uint64_t)low_part « 32) | high_part;
00057   }
00058   return value;
00059 }
```

7.17.2.14 saurion_worker_master()

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

Definition at line 677 of file low_saurion.c.

```
00677                                       {
00678   struct saurion *const s = (struct saurion *)arg;
00679   struct io_uring ring = s->rings[0];
00680   struct io_uring_cqe *cqe = NULL;
00681   struct sockaddr_in client_addr;
00682   socklen_t client_addr_len = sizeof(client_addr);
00683 
00684   add_efd(s, s->efds[0], 0);
00685   add_accept(s, &client_addr, &client_addr_len);
00686 
00687   pthread_mutex_lock(&s->status_m);
00688   s->status = 1;
00689   pthread_cond_signal(&s->status_c);
00690   pthread_mutex_unlock(&s->status_m);
00691   while (1) {
00692     int ret = io_uring_wait_cqe(&ring, &cqe);
00693     if (ret < 0) {
00694       free(cqe);
00695       return;
00696     }
00697     struct request *req = (struct request *)cqe->user_data;
00698     if (!req) {
00699       io_uring_cqe_seen(&s->rings[0], cqe);
00700       continue;
00701     }
00702     if (cqe->res < 0) {
00703       list_delete_node(&s->list, req);
00704       return;
00705     }
00706     if (req->client_socket == s->efds[0]) {
00707       io_uring_cqe_seen(&s->rings[0], cqe);
00708       list_delete_node(&s->list, req);
00709       break;
00710     }
00711     /* Mark this request as processed */
00712     io_uring_cqe_seen(&s->rings[0], cqe);
00713     switch (req->event_type) {
00714       case EVENT_TYPE_ACCEPT:
00715         handle_accept(s, cqe->res);
00716         add_accept(s, &client_addr, &client_addr_len);
00717         add_read(s, cqe->res);
00718         list_delete_node(&s->list, req);
00719         break;
00720       case EVENT_TYPE_READ:
00721         if (cqe->res < 0) {
00722           handle_error(s, req);
00723         }
00724         if (cqe->res < 1) {
00725           handle_close(s, req);
00726         }
00727         if (cqe->res > 0) {
00728           handle_read(s, req);
00729         }
00730         list_delete_node(&s->list, req);
00731         break;
00732       case EVENT_TYPE_WRITE:
00733         handle_write(s, req->client_socket);
00734         list_delete_node(&s->list, req);
00735         break;
00736     }
00737   }
00738   pthread_mutex_lock(&s->status_m);
00739   s->status = 2;
00740   pthread_cond_signal(&s->status_c);
00741   pthread_mutex_unlock(&s->status_m);
00742   return;
00743 }
```

7.17.2.15 saurion_worker_slave()

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

Definition at line 745 of file low_saurion.c.

```
00745                                       {
00746   struct saurion_wrapper *const ss = (struct saurion_wrapper *)arg;
00747   struct saurion *s = ss->s;
00748   const int sel = ss->sel;
00749   free(ss);
00750   struct io_uring ring = s->rings[sel];
00751   struct io_uring_cqe *cqe = NULL;
00752 
00753   add_efd(s, s->efds[sel], sel);
00754 
00755   pthread_mutex_lock(&s->status_m);
00756   s->status = 1;
00757   pthread_cond_signal(&s->status_c);
00758   pthread_mutex_unlock(&s->status_m);
00759   while (1) {
00760     int ret = io_uring_wait_cqe(&ring, &cqe);
00761     if (ret < 0) {
00762       free(cqe);
00763       return;
00764     }
00765     struct request *req = (struct request *)cqe->user_data;
00766     if (!req) {
00767       io_uring_cqe_seen(&ring, cqe);
00768       continue;
00769     }
00770     if (cqe->res < 0) {
00771       list_delete_node(&s->list, req);
00772       return;
00773     }
00774     if (req->client_socket == s->efds[sel]) {
00775       io_uring_cqe_seen(&ring, cqe);
00776       list_delete_node(&s->list, req);
00777       break;
00778     }
00779     /* Mark this request as processed */
00780     io_uring_cqe_seen(&ring, cqe);
00781     switch (req->event_type) {
00782       case EVENT_TYPE_READ:
00783         if (cqe->res < 0) {
00784           handle_error(s, req);
00785         }
00786         if (cqe->res < 1) {
00787           handle_close(s, req);
00788         }
00789         if (cqe->res > 0) {
00790           handle_read(s, req);
00791         }
00792         list_delete_node(&s->list, req);
00793         break;
00794       case EVENT_TYPE_WRITE:
00795         handle_write(s, req->client_socket);
00796         list_delete_node(&s->list, req);
00797         break;
00798     }
00799   }
00800   pthread_mutex_lock(&s->status_m);
00801   s->status = 2;
00802   pthread_cond_signal(&s->status_c);
00803   pthread_mutex_unlock(&s->status_m);
00804   return;
00805 }
```

7.17.3 Variable Documentation

7.17.3.1 print_mutex

```
pthread_mutex_t print_mutex
```

Definition at line 29 of file low_saurion.c.

7.18 low_saurion.c

Go to the documentation of this file.

```
#include "low_saurion.h"

#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/eventfd.h>

#define EVENT_TYPE_ACCEPT 0
#define EVENT_TYPE_READ 1
#define EVENT_TYPE_WRITE 2
#define EVENT_TYPE_WAIT 3
#define EVENT_TYPE_ERROR 4

struct request {
  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[];
};

#define MIN(a, b) ((a) < (b) ?  (a) :  (b))
#define MAX(a, b) ((a) > (b) ?  (a) :  (b))
pthread_mutex_t print_mutex;

struct saurion_wrapper {
  struct saurion *s;
  uint32_t sel;
};

static uint32_t next(struct saurion *s) {
  s->next = (s->next + 1) % s->n_threads;
  return s->next;
}

static uint64_t htonll(uint64_t value) {
  int num = 42;
  if (*(char *)&num == 42) {  // Little endian check
    uint32_t high_part = htonl((uint32_t)(value » 32));
    uint32_t low_part = htonl((uint32_t)(value & 0xFFFFFFFFLL));
    return ((uint64_t)low_part « 32) | high_part;
  }
  return value;
}

static uint64_t ntohll(uint64_t value) {
  int num = 42;
  if (*(char *)&num == 42) {  // Little endian check
    uint32_t high_part = ntohl((uint32_t)(value » 32));
    uint32_t low_part = ntohl((uint32_t)(value & 0xFFFFFFFFLL));
    return ((uint64_t)low_part « 32) | high_part;
  }
  return value;
}

void free_request(struct request *req, void **children_ptr, size_t amount) {
  if (children_ptr) {
    free(children_ptr);
    children_ptr = NULL;
  }
  for (size_t i = 0; i < amount; ++i) {
    free(req->iov[i].iov_base);
    req->iov[i].iov_base = NULL;
  }
  free(req);
  req = NULL;
  free(children_ptr);
  children_ptr = NULL;
}

[[nodiscard]]
int initialize_iovec(struct iovec *iov, size_t amount, size_t pos, const void *msg, size_t size,
                     uint8_t h) {
  if (!iov || !iov->iov_base) {
    return ERROR_CODE;
  }
  if (msg) {
    size_t len = iov->iov_len;
    char *dest = (char *)iov->iov_base;
    char *orig = (char *)msg + pos * CHUNK_SZ;
    size_t cpy_sz = 0;
    if (h) {
      if (pos == 0) {
        uint64_t send_size = htonll(size);
        memcpy(dest, &send_size, sizeof(uint64_t));
        dest += sizeof(uint64_t);
        len -= sizeof(uint64_t);
      } else {
        orig -= sizeof(uint64_t);
      }
      if ((pos + 1) == amount) {
        --len;
        cpy_sz = (len < size ?  len :  size);
        dest[cpy_sz] = 0;
      }
    }
    cpy_sz = (len < size ?  len :  size);
    memcpy(dest, orig, cpy_sz);
    dest += cpy_sz;
    size_t rem = CHUNK_SZ - (dest - (char *)iov->iov_base);
    memset(dest, 0, rem);
  } else {
    memset((char *)iov->iov_base, 0, CHUNK_SZ);
  }
  return SUCCESS_CODE;
}

[[nodiscard]]
int allocate_iovec(struct iovec *iov, size_t amount, size_t pos, size_t size, void **chd_ptr) {
  if (!iov || !chd_ptr) {
    return ERROR_CODE;
  }
  iov->iov_base = malloc(CHUNK_SZ);
  if (!iov->iov_base) {
    return ERROR_CODE;
  }
  iov->iov_len = (pos == (amount - 1) ?  (size % CHUNK_SZ) : CHUNK_SZ);
  if (iov->iov_len == 0) {
    iov->iov_len = CHUNK_SZ;
  }
  chd_ptr[pos] = iov->iov_base;
  return SUCCESS_CODE;
}

[[nodiscard]]
int set_request(struct request **r, struct Node **l, size_t s, const void *m, uint8_t h) {
  uint64_t full_size = s;
  if (h) {
    full_size += (sizeof(uint64_t) + 1);
  }
  size_t amount = full_size / CHUNK_SZ;
  amount = amount + (full_size % CHUNK_SZ == 0 ?  0 :  1);
  struct request *temp =
      (struct request *)malloc(sizeof(struct request) + sizeof(struct iovec) * amount);
  if (!temp) {
    return ERROR_CODE;
  }
  if (!*r) {
    *r = temp;
    (*r)->prev = NULL;
    (*r)->prev_size = 0;
    (*r)->prev_remain = 0;
    (*r)->next_iov = 0;
    (*r)->next_offset = 0;
  } else {
    temp->client_socket = (*r)->client_socket;
    temp->event_type = (*r)->event_type;
    temp->prev = (*r)->prev;
    temp->prev_size = (*r)->prev_size;
    temp->prev_remain = (*r)->prev_remain;
    temp->next_iov = (*r)->next_iov;
    temp->next_offset = (*r)->next_offset;
    *r = temp;
  }
  struct request *req = *r;
  req->iovec_count = (int)amount;
  void **children_ptr = (void **)malloc(amount * sizeof(void *));
  if (!children_ptr) {
    free_request(req, children_ptr, 0);
    return ERROR_CODE;
  }
  for (size_t i = 0; i < amount; ++i) {
    if (!allocate_iovec(&req->iov[i], amount, i, full_size, children_ptr)) {
      free_request(req, children_ptr, amount);
      return ERROR_CODE;
    }
    if (!initialize_iovec(&req->iov[i], amount, i, m, s, h)) {
      free_request(req, children_ptr, amount);
      return ERROR_CODE;
    }
  }
  if (list_insert(l, req, amount, children_ptr)) {
    free_request(req, children_ptr, amount);
    return ERROR_CODE;
  }
  free(children_ptr);
  return SUCCESS_CODE;
}

/*********************************** ADDERS ***********************************/
static void add_accept(struct saurion *const s, const struct sockaddr_in *const ca,
                       socklen_t *const cal) {
  int res = ERROR_CODE;
  pthread_mutex_lock(&s->m_rings[0]);
  while (res != SUCCESS_CODE) {
    struct io_uring_sqe *sqe = io_uring_get_sqe(&s->rings[0]);
    while (!sqe) {
      sqe = io_uring_get_sqe(&s->rings[0]);
      usleep(TIMEOUT_RETRY);
    }
    struct request *req = NULL;
    if (!set_request(&req, &s->list, 0, NULL, 0)) {
      free(sqe);
      usleep(TIMEOUT_RETRY);
      res = ERROR_CODE;
      continue;
    }
    req->client_socket = 0;
    req->event_type = EVENT_TYPE_ACCEPT;
    io_uring_prep_accept(sqe, s->ss, (struct sockaddr *)ca, cal, 0);
    io_uring_sqe_set_data(sqe, req);
    if (io_uring_submit(&s->rings[0]) < 0) {
      free(sqe);
      list_delete_node(&s->list, req);
      usleep(TIMEOUT_RETRY);
      res = ERROR_CODE;
      continue;
    }
    res = SUCCESS_CODE;
  }
  pthread_mutex_unlock(&s->m_rings[0]);
}

static void add_efd(struct saurion *const s, const int client_socket, int sel) {
  pthread_mutex_lock(&s->m_rings[sel]);
  int res = ERROR_CODE;
  while (res != SUCCESS_CODE) {
    struct io_uring *ring = &s->rings[sel];
    struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
    while (!sqe) {
      sqe = io_uring_get_sqe(ring);
      usleep(TIMEOUT_RETRY);
    }
    struct request *req = NULL;
    if (!set_request(&req, &s->list, CHUNK_SZ, NULL, 0)) {
      free(sqe);
      res = ERROR_CODE;
      continue;
    }
    req->event_type = EVENT_TYPE_READ;
    req->client_socket = client_socket;
    io_uring_prep_readv(sqe, client_socket, &req->iov[0], req->iovec_count, 0);
    io_uring_sqe_set_data(sqe, req);
    if (io_uring_submit(ring) < 0) {
      free(sqe);
      list_delete_node(&s->list, req);
      res = ERROR_CODE;
      continue;
    }
    res = SUCCESS_CODE;
  }
  pthread_mutex_unlock(&s->m_rings[sel]);
}

static void add_read(struct saurion *const s, const int client_socket) {
  int sel = next(s);
  int res = ERROR_CODE;
  pthread_mutex_lock(&s->m_rings[sel]);
  while (res != SUCCESS_CODE) {
    struct io_uring *ring = &s->rings[sel];
    struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
    while (!sqe) {
      sqe = io_uring_get_sqe(ring);
      usleep(TIMEOUT_RETRY);
    }
    struct request *req = NULL;
    if (!set_request(&req, &s->list, CHUNK_SZ, NULL, 0)) {
      free(sqe);
      res = ERROR_CODE;
      continue;
    }
    req->event_type = EVENT_TYPE_READ;
    req->client_socket = client_socket;
    io_uring_prep_readv(sqe, client_socket, &req->iov[0], req->iovec_count, 0);
    io_uring_sqe_set_data(sqe, req);
    if (io_uring_submit(ring) < 0) {
      free(sqe);
      list_delete_node(&s->list, req);
      res = ERROR_CODE;
      continue;
    }
    res = SUCCESS_CODE;
  }
  pthread_mutex_unlock(&s->m_rings[sel]);
}

static void add_read_continue(struct saurion *const s, struct request *oreq, const int sel) {
  pthread_mutex_lock(&s->m_rings[sel]);
  int res = ERROR_CODE;
  while (res != SUCCESS_CODE) {
    struct io_uring *ring = &s->rings[sel];
    struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
    while (!sqe) {
      sqe = io_uring_get_sqe(ring);
      usleep(TIMEOUT_RETRY);
    }
    if (!set_request(&oreq, &s->list, oreq->prev_remain, NULL, 0)) {
      free(sqe);
      res = ERROR_CODE;
      continue;
    }
    io_uring_prep_readv(sqe, oreq->client_socket, &oreq->iov[0], oreq->iovec_count, 0);
    io_uring_sqe_set_data(sqe, oreq);
    if (io_uring_submit(ring) < 0) {
      free(sqe);
      list_delete_node(&s->list, oreq);
      res = ERROR_CODE;
      continue;
    }
    res = SUCCESS_CODE;
  }
  pthread_mutex_unlock(&s->m_rings[sel]);
}

static void add_write(struct saurion *const s, int fd, const char *const str, const int sel) {
  int res = ERROR_CODE;
  pthread_mutex_lock(&s->m_rings[sel]);
  while (res != SUCCESS_CODE) {
    struct io_uring *ring = &s->rings[sel];
    struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
    while (!sqe) {
      sqe = io_uring_get_sqe(ring);
      usleep(TIMEOUT_RETRY);
    }
    struct request *req = NULL;
    if (!set_request(&req, &s->list, strlen(str), (void *)str, 1)) {
      free(sqe);
      res = ERROR_CODE;
      continue;
    }
    req->event_type = EVENT_TYPE_WRITE;
    req->client_socket = fd;
    io_uring_prep_writev(sqe, req->client_socket, req->iov, req->iovec_count, 0);
    io_uring_sqe_set_data(sqe, req);
    if (io_uring_submit(ring) < 0) {
      free(sqe);
      list_delete_node(&s->list, req);
      res = ERROR_CODE;
      usleep(TIMEOUT_RETRY);
      continue;
    }
    res = SUCCESS_CODE;
  }
  pthread_mutex_unlock(&s->m_rings[sel]);
}

/*********************************** HANDLERS ***********************************/
static void handle_accept(const struct saurion *const s, const int fd) {
  if (s->cb.on_connected) {
    s->cb.on_connected(fd, s->cb.on_connected_arg);
  }
}

[[nodiscard]]
int read_chunk(void **dest, size_t *len, struct request *const req) {
  // Initial checks
  if (req->iovec_count == 0) {
    return ERROR_CODE;
  }

  // Initization
  size_t max_iov_cont = 0;  //< Total size of request
  for (size_t i = 0; i < req->iovec_count; ++i) {
    max_iov_cont += req->iov[i].iov_len;
  }
  size_t cont_sz = 0;       //< Message content size
  size_t cont_rem = 0;      //< Remaining bytes of message content
  size_t curr_iov = 0;      //< IOVEC num currently reading
  size_t curr_iov_off = 0;  //< Offset in bytes of the current IOVEC
  size_t dest_off = 0;      //< Write offset on the destiny array
  void *dest_ptr = NULL;    //< Destiny pointer, could be dest or prev
  if (req->prev && req->prev_size && req->prev_remain) {
    // There's a previous unfinished message
    cont_sz = req->prev_size;
    cont_rem = req->prev_remain;
    curr_iov = 0;
    curr_iov_off = 0;
    dest_off = cont_sz - cont_rem;
    if (cont_rem <= max_iov_cont) {
      *dest = req->prev;
      dest_ptr = *dest;
      req->prev = NULL;
      req->prev_size = 0;
      req->prev_remain = 0;
    } else {
      dest_ptr = req->prev;
      *dest = NULL;
    }
  } else if (req->next_iov || req->next_offset) {
    // Reading the next message
    curr_iov = req->next_iov;
    curr_iov_off = req->next_offset;
    cont_sz = *((size_t *)(req->iov[curr_iov].iov_base + curr_iov_off));
    cont_sz = ntohll(cont_sz);
    curr_iov_off += sizeof(uint64_t);
    cont_rem = cont_sz;
    dest_off = cont_sz - cont_rem;
    if ((curr_iov_off + cont_rem + 1) <= max_iov_cont) {
      *dest = malloc(cont_sz);
      dest_ptr = *dest;
    } else {
      req->prev = malloc(cont_sz);
      dest_ptr = req->prev;
      *dest = NULL;
      *len = 0;
    }
  } else {
    // Reading the first message
    curr_iov = 0;
    curr_iov_off = 0;
    cont_sz = *((size_t *)(req->iov[curr_iov].iov_base + curr_iov_off));
    cont_sz = ntohll(cont_sz);
    curr_iov_off += sizeof(uint64_t);
    cont_rem = cont_sz;
    dest_off = cont_sz - cont_rem;
    if (cont_rem <= max_iov_cont) {
      *dest = malloc(cont_sz);
      dest_ptr = *dest;
    } else {
      req->prev = malloc(cont_sz);
      dest_ptr = req->prev;
      *dest = NULL;
    }
  }
  size_t curr_iov_msg_rem = 0;

  // Copy loop
  uint8_t ok = 1UL;
  while (1) {
    curr_iov_msg_rem = MIN(cont_rem, (req->iov[curr_iov].iov_len - curr_iov_off));
    memcpy(dest_ptr + dest_off, req->iov[curr_iov].iov_base + curr_iov_off, curr_iov_msg_rem);
    dest_off += curr_iov_msg_rem;
    curr_iov_off += curr_iov_msg_rem;
    cont_rem -= curr_iov_msg_rem;
    if (cont_rem <= 0) {
      // Finish reading
      if (*((uint8_t *)(req->iov[curr_iov].iov_base + curr_iov_off)) != 0) {
        ok = 0UL;
      }
      *len = cont_sz;
      ++curr_iov_off;
      break;
    }
    if (curr_iov_off >= (req->iov[curr_iov].iov_len)) {
      ++curr_iov;
      if (curr_iov == req->iovec_count) {
        break;
      }
      curr_iov_off = 0;
    }
  }

  // Update status
  if (req->prev) {
    req->prev_size = cont_sz;
    req->prev_remain = cont_rem;
    *dest = NULL;
    len = 0;
  } else {
    req->prev_size = 0;
    req->prev_remain = 0;
  }
  if (curr_iov < req->iovec_count) {
    uint64_t next_sz = *(uint64_t *)(req->iov[curr_iov].iov_base + curr_iov_off);
    if ((req->iov[curr_iov].iov_len > curr_iov_off) && next_sz) {
      req->next_iov = curr_iov;
      req->next_offset = curr_iov_off;
    } else {
      req->next_iov = 0;
      req->next_offset = 0;
    }
  }

  // Finish
  if (!ok) {
    // Esto solo es posible si no se encuentra un 0 al final del la lectura
    // buscar el siguiente 0 y ...  probar fortuna
    free(dest_ptr);
    dest_ptr = NULL;
    *dest = NULL;
    *len = 0;
    req->next_iov = 0;
    req->next_offset = 0;
    for (size_t i = curr_iov; i < req->iovec_count; ++i) {
      for (size_t j = curr_iov_off; j < req->iov[i].iov_len; ++j) {
        uint8_t foot = *(uint8_t *)(req->iov[i].iov_base + j);
        if (foot == 0) {
          req->next_iov = i;
          req->next_offset = (j + 1) % req->iov[i].iov_len;
          return ERROR_CODE;
        }
      }
    }
    return ERROR_CODE;
  }
  return SUCCESS_CODE;
}

static void handle_read(struct saurion *const s, struct request *const req) {
  void *msg = NULL;
  size_t len = 0;
  while (1) {
    if (!read_chunk(&msg, &len, req)) {
      break;
    }
    // Hay siguiente
    if (req->next_iov || req->next_offset) {
      if (s->cb.on_readed && msg) {
        s->cb.on_readed(req->client_socket, msg, len, s->cb.on_readed_arg);
      }
      free(msg);
      msg = NULL;
      continue;
    }
    // Hay previo pero no se ha completado
    if (req->prev && req->prev_size && req->prev_remain) {
      add_read_continue(s, req, next(s));
      return;
    }
    // Hay un único mensaje y se ha completado
    if (s->cb.on_readed && msg) {
      s->cb.on_readed(req->client_socket, msg, len, s->cb.on_readed_arg);
    }
    free(msg);
    msg = NULL;
    break;
  }
  add_read(s, req->client_socket);
}

static void handle_write(const struct saurion *const s, const int fd) {
  if (s->cb.on_wrote) {
    s->cb.on_wrote(fd, s->cb.on_wrote_arg);
  }
}

static void handle_error(const struct saurion *const s, const struct request *const req) {
  if (s->cb.on_error) {
    const char *resp = "ERROR";
    s->cb.on_error(req->client_socket, resp, (ssize_t)strlen(resp), s->cb.on_error_arg);
  }
}

static void handle_close(const struct saurion *const s, const struct request *const req) {
  if (s->cb.on_closed) {
    s->cb.on_closed(req->client_socket, s->cb.on_closed_arg);
  }
  close(req->client_socket);
}

/************************************ INTERFACE ************************************/
int EXTERNAL_set_socket(const int p) {
  int sock = 0;
  struct sockaddr_in srv_addr;

  sock = socket(PF_INET, SOCK_STREAM, 0);
  if (sock < 1) {
    return ERROR_CODE;
  }

  int enable = 1;
  if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(int)) < 0) {
    return ERROR_CODE;
  }
  if (setsockopt(sock, SOL_SOCKET, SO_REUSEPORT, &enable, sizeof(int)) < 0) {
    return ERROR_CODE;
  }

  memset(&srv_addr, 0, sizeof(srv_addr));
  srv_addr.sin_family = AF_INET;
  srv_addr.sin_port = htons(p);
  srv_addr.sin_addr.s_addr = htonl(INADDR_ANY);

  /* We bind to a port and turn this socket into a listening
* socket.
* */
  if (bind(sock, (const struct sockaddr *)&srv_addr, sizeof(srv_addr)) < 0) {
    return ERROR_CODE;
  }

  if (listen(sock, ACCEPT_QUEUE) < 0) {
    return ERROR_CODE;
  }

  return sock;
}

[[nodiscard]]
struct saurion *saurion_create(uint32_t n_threads) {
  // Asignar memoria
  struct saurion *p = (struct saurion *)malloc(sizeof(struct saurion));
  if (!p) {
    return NULL;
  }
  // Inicializar mutex
  int ret = 0;
  ret = pthread_mutex_init(&p->status_m, NULL);
  if (ret) {
    free(p);
    return NULL;
  }
  ret = pthread_cond_init(&p->status_c, NULL);
  if (ret) {
    free(p);
    return NULL;
  }
  p->m_rings = (pthread_mutex_t *)malloc(n_threads * sizeof(pthread_mutex_t));
  if (!p->m_rings) {
    free(p);
    return NULL;
  }
  for (uint32_t i = 0; i < n_threads; ++i) {
    pthread_mutex_init(&(p->m_rings[i]), NULL);
  }
  // Inicializar miembros
  p->ss = 0;
  n_threads = (n_threads < 2 ?  2 :  n_threads);
  n_threads = (n_threads > NUM_CORES ? NUM_CORES : n_threads);
  p->n_threads = n_threads;
  p->status = 0;
  p->list = NULL;
  p->cb.on_connected = NULL;
  p->cb.on_connected_arg = NULL;
  p->cb.on_readed = NULL;
  p->cb.on_readed_arg = NULL;
  p->cb.on_wrote = NULL;
  p->cb.on_wrote_arg = NULL;
  p->cb.on_closed = NULL;
  p->cb.on_closed_arg = NULL;
  p->cb.on_error = NULL;
  p->cb.on_error_arg = NULL;
  p->next = 0;
  // Inicializar efds
  p->efds = (int *)malloc(sizeof(int) * p->n_threads);
  if (!p->efds) {
    free(p->m_rings);
    free(p);
    return NULL;
  }
  for (uint32_t i = 0; i < p->n_threads; ++i) {
    p->efds[i] = eventfd(0, EFD_NONBLOCK);
    if (p->efds[i] == ERROR_CODE) {
      for (uint32_t j = 0; j < i; ++j) {
        close(p->efds[j]);
      }
      free(p->efds);
      free(p->m_rings);
      free(p);
      return NULL;
    }
  }
  // Inicializar rings
  p->rings = (struct io_uring *)malloc(sizeof(struct io_uring) * p->n_threads);
  if (!p->rings) {
    for (uint32_t j = 0; j < p->n_threads; ++j) {
      close(p->efds[j]);
    }
    free(p->efds);
    free(p->m_rings);
    free(p);
    return NULL;
  }
  for (uint32_t i = 0; i < p->n_threads; ++i) {
    memset(&p->rings[i], 0, sizeof(struct io_uring));
    ret = io_uring_queue_init(SAURION_RING_SIZE, &p->rings[i], 0);
    if (ret) {
      for (uint32_t j = 0; j < p->n_threads; ++j) {
        close(p->efds[j]);
      }
      free(p->efds);
      free(p->rings);
      free(p->m_rings);
      free(p);
      return NULL;
    }
  }
  p->pool = ThreadPool_create(p->n_threads);
  return p;
}

void saurion_worker_master(void *arg) {
  struct saurion *const s = (struct saurion *)arg;
  struct io_uring ring = s->rings[0];
  struct io_uring_cqe *cqe = NULL;
  struct sockaddr_in client_addr;
  socklen_t client_addr_len = sizeof(client_addr);

  add_efd(s, s->efds[0], 0);
  add_accept(s, &client_addr, &client_addr_len);

  pthread_mutex_lock(&s->status_m);
  s->status = 1;
  pthread_cond_signal(&s->status_c);
  pthread_mutex_unlock(&s->status_m);
  while (1) {
    int ret = io_uring_wait_cqe(&ring, &cqe);
    if (ret < 0) {
      free(cqe);
      return;
    }
    struct request *req = (struct request *)cqe->user_data;
    if (!req) {
      io_uring_cqe_seen(&s->rings[0], cqe);
      continue;
    }
    if (cqe->res < 0) {
      list_delete_node(&s->list, req);
      return;
    }
    if (req->client_socket == s->efds[0]) {
      io_uring_cqe_seen(&s->rings[0], cqe);
      list_delete_node(&s->list, req);
      break;
    }
    /* Mark this request as processed */
    io_uring_cqe_seen(&s->rings[0], cqe);
    switch (req->event_type) {
      case EVENT_TYPE_ACCEPT:
        handle_accept(s, cqe->res);
        add_accept(s, &client_addr, &client_addr_len);
        add_read(s, cqe->res);
        list_delete_node(&s->list, req);
        break;
      case EVENT_TYPE_READ:
        if (cqe->res < 0) {
          handle_error(s, req);
        }
        if (cqe->res < 1) {
          handle_close(s, req);
        }
        if (cqe->res > 0) {
          handle_read(s, req);
        }
        list_delete_node(&s->list, req);
        break;
      case EVENT_TYPE_WRITE:
        handle_write(s, req->client_socket);
        list_delete_node(&s->list, req);
        break;
    }
  }
  pthread_mutex_lock(&s->status_m);
  s->status = 2;
  pthread_cond_signal(&s->status_c);
  pthread_mutex_unlock(&s->status_m);
  return;
}

void saurion_worker_slave(void *arg) {
  struct saurion_wrapper *const ss = (struct saurion_wrapper *)arg;
  struct saurion *s = ss->s;
  const int sel = ss->sel;
  free(ss);
  struct io_uring ring = s->rings[sel];
  struct io_uring_cqe *cqe = NULL;

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

  pthread_mutex_lock(&s->status_m);
  s->status = 1;
  pthread_cond_signal(&s->status_c);
  pthread_mutex_unlock(&s->status_m);
  while (1) {
    int ret = io_uring_wait_cqe(&ring, &cqe);
    if (ret < 0) {
      free(cqe);
      return;
    }
    struct request *req = (struct request *)cqe->user_data;
    if (!req) {
      io_uring_cqe_seen(&ring, cqe);
      continue;
    }
    if (cqe->res < 0) {
      list_delete_node(&s->list, req);
      return;
    }
    if (req->client_socket == s->efds[sel]) {
      io_uring_cqe_seen(&ring, cqe);
      list_delete_node(&s->list, req);
      break;
    }
    /* Mark this request as processed */
    io_uring_cqe_seen(&ring, cqe);
    switch (req->event_type) {
      case EVENT_TYPE_READ:
        if (cqe->res < 0) {
          handle_error(s, req);
        }
        if (cqe->res < 1) {
          handle_close(s, req);
        }
        if (cqe->res > 0) {
          handle_read(s, req);
        }
        list_delete_node(&s->list, req);
        break;
      case EVENT_TYPE_WRITE:
        handle_write(s, req->client_socket);
        list_delete_node(&s->list, req);
        break;
    }
  }
  pthread_mutex_lock(&s->status_m);
  s->status = 2;
  pthread_cond_signal(&s->status_c);
  pthread_mutex_unlock(&s->status_m);
  return;
}

[[nodiscard]]
int saurion_start(struct saurion *const s) {
  pthread_mutex_init(&print_mutex, NULL);
  ThreadPool_init(s->pool);
  ThreadPool_add_default(s->pool, saurion_worker_master, s);
  struct saurion_wrapper *ss = NULL;
  for (uint32_t i = 1; i < s->n_threads; ++i) {
    ss = (struct saurion_wrapper *)malloc(sizeof(struct saurion_wrapper));
    ss->s = s;
    ss->sel = i;
    ThreadPool_add_default(s->pool, saurion_worker_slave, ss);
  }
  return SUCCESS_CODE;
}

void saurion_stop(const struct saurion *const s) {
  uint64_t u = 1;
  for (uint32_t i = 0; i < s->n_threads; ++i) {
    while (write(s->efds[i], &u, sizeof(u)) < 0) {
      usleep(TIMEOUT_RETRY);
    }
  }
  ThreadPool_wait_empty(s->pool);
}

void saurion_destroy(struct saurion *const s) {
  pthread_mutex_lock(&s->status_m);
  while (s->status == 1) {
    pthread_cond_wait(&s->status_c, &s->status_m);
  }
  pthread_mutex_unlock(&s->status_m);
  ThreadPool_destroy(s->pool);
  for (uint32_t i = 0; i < s->n_threads; ++i) {
    io_uring_queue_exit(&s->rings[i]);
    pthread_mutex_destroy(&s->m_rings[i]);
  }
  free(s->m_rings);
  list_free(&s->list);
  for (uint32_t i = 0; i < s->n_threads; ++i) {
    close(s->efds[i]);
  }
  free(s->efds);
  if (!s->ss) {
    close(s->ss);
  }
  free(s->rings);
  pthread_mutex_destroy(&s->status_m);
  pthread_cond_destroy(&s->status_c);
  free(s);
}

void saurion_send(struct saurion *const s, const int fd, const char *const msg) {
  add_write(s, fd, msg, next(s));
}
```

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

```
#include "saurion.hpp"
#include <cstdint>
#include "low_saurion.h"
```

Include dependency graph for saurion.cpp:

7.20 saurion.cpp

Go to the documentation of this file.

```
#include "saurion.hpp"

#include <cstdint>

#include "low_saurion.h"

Saurion::Saurion(const uint32_t thds, const int sck) noexcept {
  this->s = saurion_create(thds);
  if (!this->s) {
    return;
  }
  this->s->ss = sck;
}

Saurion::~Saurion() { saurion_destroy(this->s); }

void Saurion::init() noexcept {
  if (!saurion_start(this->s)) {
    return;
  }
}

void Saurion::stop() noexcept { saurion_stop(this->s); }

Saurion *Saurion::on_connected(Saurion::ConnectedCb ncb, void *arg) noexcept {
  s->cb.on_connected = ncb;
  s->cb.on_connected_arg = arg;
  return this;
}

Saurion *Saurion::on_readed(Saurion::ReadedCb ncb, void *arg) noexcept {
  s->cb.on_readed = ncb;
  s->cb.on_readed_arg = arg;
  return this;
}

Saurion *Saurion::on_wrote(Saurion::WroteCb ncb, void *arg) noexcept {
  s->cb.on_wrote = ncb;
  s->cb.on_wrote_arg = arg;
  return this;
}

Saurion *Saurion::on_closed(Saurion::ClosedCb ncb, void *arg) noexcept {
  s->cb.on_closed = ncb;
  s->cb.on_closed_arg = arg;
  return this;
}

Saurion *Saurion::on_error(Saurion::ErrorCb ncb, void *arg) noexcept {
  s->cb.on_error = ncb;
  s->cb.on_error_arg = arg;
  return this;
}

void Saurion::send(const int fd, const char *const msg) noexcept { saurion_send(this->s, fd, msg); }
```

7.21 /__w/saurion/saurion/src/threadpool.cpp File Reference

```
#include "threadpool.hpp"
#include <pthread.h>
#include <stdexcept>
```

Include dependency graph for threadpool.cpp:

Typedefs

- using TP = ThreadPool
- using T = Task

7.21.1 Typedef Documentation

7.21.1.1 T

```
using T = Task
```

Definition at line 12 of file threadpool.cpp.

7.21.1.2 TP

```
using TP = ThreadPool
```

Definition at line 9 of file threadpool.cpp.

7.22 threadpool.cpp

Go to the documentation of this file.

```
00001 // This is a personal academic project.  Dear PVS-Studio, please check it.
00002 // PVS-Studio Static Code Analyzer for C, C++, C#, and Java:  https://pvs-studio.com
00003 #include "threadpool.hpp"
00004
00005 #include <pthread.h>
00006
00007 #include <stdexcept>
00008
00009 using TP = ThreadPool;
00010
00011 //********* Task ************
00012 using T = Task;
00013 T::Task(void (*nfn)(void*), void* narg) :  function(nfn), argument(narg) {}
00014
00015 //********* AsyncQueue ************
00016 TP::AsyncQueue::AsyncQueue(uint32_t cnt) :  m_max(cnt), m_cnt(0) {}
00017
00018 TP::AsyncQueue::~AsyncQueue() { pthread_mutex_destroy(&m_mtx); }
00019
00020 void TP::AsyncQueue::push(Task* task) {
00021   pthread_mutex_lock(&m_mtx);
00022   m_queue.push(task);
00023   pthread_mutex_unlock(&m_mtx);
00024 }
00025 Task* TP::AsyncQueue::front() {
00026   pthread_mutex_lock(&m_mtx);
00027   if ((m_cnt >= m_max) && (m_max != 0)) {
00028     pthread_mutex_unlock(&m_mtx);
00029     throw std::out_of_range("reached max parallel tasks");
00030   }
00031   Task* task = m_queue.front();
00032   m_queue.pop();
00033   ++m_cnt;
00034   pthread_mutex_unlock(&m_mtx);
00035   return task;
00036 }
00037 void TP::AsyncQueue::pop() {
  pthread_mutex_lock(&m_mtx);
  --m_cnt;
  pthread_mutex_unlock(&m_mtx);
}

bool TP::AsyncQueue::empty() {
  pthread_mutex_lock(&m_mtx);
  bool empty = m_queue.empty();
  pthread_mutex_unlock(&m_mtx);
  return empty;
}

//********* AsyncMultiQueue ************

TP::AsyncMultiQueue::AsyncMultiQueue() {
  new_queue(0, 0);
  m_it = m_queues.begin();
}
TP::AsyncMultiQueue::~AsyncMultiQueue() {
  for (auto& queue :  m_queues) {
    delete queue.second;
  }
}

void TP::AsyncMultiQueue::new_queue(uint32_t qid, uint32_t cnt) {
  if (m_queues.find(qid) != m_queues.end()) {
    throw std::out_of_range("queue already exists");
  }
  m_queues.emplace(qid, new TP::AsyncQueue(cnt));
}
void TP::AsyncMultiQueue::remove_queue(uint32_t qid) {
  auto queue = m_queues.find(qid);
  if (queue == m_queues.end()) {
    throw std::out_of_range("queue not found");
  }
  delete queue->second;
  m_queues.erase(qid);
}

void TP::AsyncMultiQueue::push(uint32_t qid, void (*nfn)(void*), void* arg) {
  m_queues.at(qid)->push(new Task{nfn, arg});
}
Task* TP::AsyncMultiQueue::front(uint32_t& qid) {
  if (empty()) {
    throw std::out_of_range("empty queue");
  }
  auto newit = m_it;
  ++newit;
  while (newit != m_it) {
    if (newit == m_queues.end()) {
      newit = m_queues.begin();
    }
    if (!newit->second->empty()) {
      break;
    }
    ++newit;
  }
  Task* task = newit->second->front();
  qid = newit->first;
  m_it = newit;
  return task;
}
void TP::AsyncMultiQueue::pop(uint32_t qid) { m_queues.at(qid)->pop(); }
void TP::AsyncMultiQueue::clear() {
  if (empty()) {
    return;
  }
  auto newit = m_queues.begin();
  while (newit != m_queues.end()) {
    while (!newit->second->empty()) {
      delete newit->second->front();
      newit->second->pop();
    }
    ++newit;
  }
}

bool TP::AsyncMultiQueue::empty() {
  for (auto& queue :  m_queues) {
    if (!queue.second->empty()) {
      return false;
    }
  }
  return true;
}

//********* ThreadPool ************
```

```

pthread_mutex_t TP::s_mtx = PTHREAD_MUTEX_INITIALIZER;

TP::ThreadPool() :  ThreadPool(4) {}

TP::ThreadPool(size_t num_threads)
    :  m_nth(num_threads < 2 ?  2 :  num_threads),
      m_started(0),
      m_ths(new pthread_t[m_nth]{0}),
      m_fstop(0),
      m_faccept(0) {}

void TP::init() {
  if (m_started != 0) {
    return;
  }
  m_faccept = 1;
  m_fstop = 0;
  pthread_mutex_lock(&s_mtx);
  for (size_t i = 0; i < m_nth; ++i) {
    pthread_create(&m_ths[i], nullptr, thread_entry, this);
    ++m_started;
  }
  pthread_mutex_unlock(&s_mtx);
}
void TP::stop() {
  if (m_started == 0) {
    return;
  }
  m_fstop = 0;
  m_faccept = 0;
  wait_empty();

  pthread_mutex_lock(&m_q_mtx);
  m_fstop = 1;
  pthread_cond_broadcast(&m_q_cond);
  pthread_mutex_unlock(&m_q_mtx);
  // Detener los hilos
  for (size_t i = 0; i < m_started; ++i) {
    pthread_join(m_ths[i], nullptr);
  }
  m_started = 0;
  m_nth = 0;
}
void TP::add(uint32_t qid, void (*nfn)(void*), void* arg) {
  if (m_faccept == 0) {
    throw std::logic_error("threadpool already closed");
  }
  if (nfn == nullptr) {
    throw std::logic_error("function pointer cannot be null");
  }
  bool failed = false;
  pthread_mutex_lock(&m_q_mtx);
  try {
    m_queues.push(qid, nfn, arg);
  } catch (const std::out_of_range& e) {
    failed = true;
  }
  pthread_cond_signal(&m_q_cond);
  pthread_mutex_unlock(&m_q_mtx);
  if (failed) {
    throw std::out_of_range("queue not found");
  }
}
void TP::add(void (*nfn)(void*), void* arg) {
  add(0, nfn, arg);  // Agregar a la cola por defecto
}
void TP::new_queue(uint32_t qid, uint32_t cnt) {
  pthread_mutex_lock(&m_q_mtx);
  try {
    m_queues.new_queue(qid, cnt);
  } catch (const std::out_of_range& e) {
    pthread_mutex_unlock(&m_q_mtx);
    throw e;
  }
  pthread_mutex_unlock(&m_q_mtx);
}
void TP::remove_queue(uint32_t qid) {
  if (qid != 0) {
    bool failed = false;
    pthread_mutex_lock(&m_q_mtx);
    try {
      m_queues.remove_queue(qid);
    } catch (const std::out_of_range& e) {
      failed = true;
    }
    pthread_cond_signal(&m_q_cond);
    pthread_mutex_unlock(&m_q_mtx);
    if (failed) {
      throw std::out_of_range("queue not found");
    }
  }
}
bool TP::empty() { return m_queues.empty(); }
void TP::wait_closeable() {
  pthread_mutex_lock(&m_q_mtx);
  while (empty() && m_fstop == 0) {
    pthread_cond_wait(&m_q_cond, &m_q_mtx);
  }
  pthread_mutex_unlock(&m_q_mtx);
}
void TP::wait_empty() {
  pthread_mutex_lock(&m_q_mtx);
  while (!m_queues.empty()) {
    pthread_cond_wait(&m_q_cond, &m_q_mtx);
  }
  pthread_mutex_unlock(&m_q_mtx);
}
TP::~ThreadPool() {
  stop();
  m_queues.clear();
  delete[] m_ths;
  pthread_mutex_destroy(&s_mtx);
}

void TP::thread_worker() {
  // Lógica del trabajador del hilo
  uint32_t qid = 0;
  while (m_fstop == 0) {
    // Buscar una tarea para ejecutar
    try {
      pthread_mutex_lock(&m_q_mtx);
      Task* task = m_queues.front(qid);
      pthread_cond_signal(&m_q_cond);
      pthread_mutex_unlock(&m_q_mtx);
      try {
        task->function(task->argument);
      } catch (...)  {
      }
      delete task;  // TODO delete task
      pthread_mutex_lock(&m_q_mtx);
      m_queues.pop(qid);
      pthread_cond_broadcast(&m_q_cond);
      pthread_mutex_unlock(&m_q_mtx);
    } catch (const std::out_of_range& e) {
      pthread_mutex_unlock(&m_q_mtx);
      wait_closeable();
    }
  }
}
void* TP::thread_entry(void* arg) {
  auto* pool = static_cast<TP*>(arg);
  pool->thread_worker();
  return nullptr;
}
```

Index