Saurion

Generated by Doxygen 1.9.4

Chapter 1

Todo List

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

add message contraint

validar `msg_size`, crear maximos

validar `offsets`

Chapter 2

Module Index

2.1 Modules

Here is a list of all modules:

Chapter 3

Class Index

3.1 Class List

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

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Module Documentation

5.1 LowSaurion

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

Classes

- struct saurion

  Main structure for managing io_uring and socket events.

Macros

- #define _POSIX_C_SOURCE 200809L
- #define PACKING_SZ 32

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

Functions

- int saurion_set_socket (int p)

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

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

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

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

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

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

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

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

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

5.1.1 Detailed Description

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

This function allocates memory for each `struct iovec`

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

The messages are composed of three main parts:

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

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

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

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

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

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

The structure of the message is as follows:

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

The structure of the `iovec` division is:

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

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

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

Basic usage example:

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

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

Author

Israel

Date

2024

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

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

Parameters

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

Return values

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

Note

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

5.1.2 Macro Definition Documentation

5.1.2.1 _POSIX_C_SOURCE

```
#define _POSIX_C_SOURCE 200809L
```

Definition at line 107 of file low_saurion.h.

5.1.2.2 PACKING_SZ

```
#define PACKING_SZ 32
```

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

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

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

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

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

Definition at line 139 of file low_saurion.h.

5.1.3 Function Documentation

5.1.3.1 allocate_iovec()

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

Definition at line 159 of file low_saurion.c.

```
{
  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;
}
```

5.1.3.2 free_request()

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

Definition at line 91 of file low_saurion.c.

```
{
  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;
}
```

5.1.3.3 initialize_iovec()

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

Initializes a specified iovec structure with a message fragment.

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

Parameters

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

Return values

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

Note

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

Attention

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

Warning

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

Definition at line 111 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;
}
```

5.1.3.4 read_chunk()

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

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

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

Parameters

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

Note

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

Warning

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

Returns

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

Return values

SUCCESS_CODE	No malformed message found.
ERROR_CODE	Malformed message found.

Todo add message contraint

validar `msg_size`, crear maximos

validar `offsets`

Definition at line 428 of file low_saurion.c.

```
{
  if (req->iovec_count == 0)
    {
      return ERROR_CODE;
    }

  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;
  size_t cont_rem = 0;
  size_t curr_iov = 0;
  size_t curr_iov_off = 0;
  size_t dest_off = 0;
  void *dest_ptr = NULL;
  if (req->prev && req->prev_size && req->prev_remain)
    {
      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)
    {
      curr_iov = req->next_iov;
      curr_iov_off = req->next_offset;
      cont_sz = *(
          (size_t *)(((uint8_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
    {
      curr_iov = 0;
      curr_iov_off = 0;
      cont_sz = *(
          (size_t *)(((uint8_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;

  uint8_t ok = 1UL;
  while (1)
    {
      curr_iov_msg_rem
          = MIN (cont_rem, (req->iov[curr_iov].iov_len - curr_iov_off));
      memcpy ((uint8_t *)dest_ptr + dest_off,
              ((uint8_t *)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)
        {
          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;
        }
    }

  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 *)(((uint8_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;
        }
    }

  if (ok)
    {
      return SUCCESS_CODE;
    }
  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;
}
```

5.1.3.5 saurion_create()

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

Creates an instance of the `saurion` structure.

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

Parameters

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

Returns

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

Definition at line 707 of file low_saurion.c.

```
{
  LOG_INIT (" ");
  struct saurion *p = (struct saurion *)malloc (sizeof (struct saurion));
  if (!p)
    {
      LOG_END (" ");
      return NULL;
    }
  int ret = 0;
  ret = pthread_mutex_init (&p->status_m, NULL);
  if (ret)
    {
      free (p);
      LOG_END (" ");
      return NULL;
    }
  ret = pthread_cond_init (&p->status_c, NULL);
  if (ret)
    {
      free (p);
      LOG_END (" ");
      return NULL;
    }
  p->m_rings
      = (pthread_mutex_t *)malloc (n_threads * sizeof (pthread_mutex_t));
  if (!p->m_rings)
    {
      free (p);
      LOG_END (" ");
      return NULL;
    }
  for (uint32_t i = 0; i < n_threads; ++i)
    {
      pthread_mutex_init (&(p->m_rings[i]), NULL);
    }
  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;
  p->efds = (int *)malloc (sizeof (int) * p->n_threads);
  if (!p->efds)
    {
      free (p->m_rings);
      free (p);
      LOG_END (" ");
      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);
          LOG_END (" ");
          return NULL;
        }
    }
  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);
      LOG_END (" ");
      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);
          LOG_END (" ");
          return NULL;
        }
    }
  p->pool = threadpool_create (p->n_threads);
  LOG_END (" ");
  return p;
}
```

5.1.3.6 saurion_destroy()

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

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

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

Parameters

s	Pointer to the `saurion` structure.

Definition at line 1059 of file low_saurion.c.

```
{
  pthread_mutex_lock (&s->status_m);
  while (s->status > 0)
    {
      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);
}
```

5.1.3.7 saurion_send()

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

Sends a message through a socket using io_uring.

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

Parameters

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

Definition at line 1091 of file low_saurion.c.

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

5.1.3.8 saurion_set_socket()

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

Creates a socket.

Creates and sets a socket, ready for saurion configuration.

Parameters

p	port

Returns

result of socket creation.

Definition at line 670 of file low_saurion.c.

```
{
  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;
    }

  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);

  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;
}
```

5.1.3.9 saurion_start()

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

Starts event processing in the `saurion` structure.

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

Parameters

s	Pointer to the `saurion` structure.

Returns

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

Definition at line 1018 of file low_saurion.c.

```
{
  pthread_mutex_init (&print_mutex, NULL);
  threadpool_init (s->pool);
  threadpool_add (s->pool, saurion_worker_master, s);
  struct saurion_wrapper *ss = NULL;
  for (uint32_t i = 1; i < s->n_threads; ++i)
    {
      ss = (struct saurion_wrapper *)malloc (sizeof (struct saurion_wrapper));
      if (!ss)
        {
          return ERROR_CODE;
        }
      ss->s = s;
      ss->sel = i;
      threadpool_add (s->pool, saurion_worker_slave, ss);
    }
  pthread_mutex_lock (&s->status_m);
  while (s->status < (int)s->n_threads)
    {
      pthread_cond_wait (&s->status_c, &s->status_m);
    }
  pthread_mutex_unlock (&s->status_m);
  return SUCCESS_CODE;
}
```

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 1045 of file low_saurion.c.

```
{
  uint64_t u = 1;
  for (uint32_t i = 0; i < s->n_threads; ++i)
    {
      while (write (s->efds[i], &u, sizeof (u)) < 0)
        {
          nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
        }
    }
  threadpool_wait_empty (s->pool);
}
```

5.1.3.11 set_request()

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

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

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

Parameters

r	Pointer to a pointer to the request structure. If NULL, a new request is created.
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 182 of file low_saurion.c.

```
{
  uint64_t full_size = s;
  if (h)
    {
      full_size += (sizeof (uint64_t) + sizeof (uint8_t));
    }
  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;
}
```

5.2 ThreadPool

Functions

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

5.2.1 Detailed Description

5.2.2 Function Documentation

5.2.2.1 threadpool_add()

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

Definition at line 175 of file threadpool.c.

```
00177 {
00178   LOG_INIT (" ");
00179   if (pool == NULL || function == NULL)
00180     {
00181       LOG_END (" ");
00182       return;
00183     }
00184
00185   struct task *new_task = malloc (sizeof (struct task));
00186   if (new_task == NULL)
00187     {
00188       perror ("Failed to allocate task");
00189       LOG_END (" ");
00190       return;
00191     }
00192
00193   new_task->function = function;
00194   new_task->argument = argument;
00195   new_task->next = NULL;
00196
00197   pthread_mutex_lock (&pool->queue_lock);
00198
00199   if (pool->task_queue_head == NULL)
00200     {
00201       pool->task_queue_head = new_task;
00202       pool->task_queue_tail = new_task;
00203     }
00204   else
00205     {
00206       pool->task_queue_tail->next = new_task;
00207       pool->task_queue_tail = new_task;
00208     }
00209   pthread_cond_signal (&pool->queue_cond);
00210
00211   pthread_mutex_unlock (&pool->queue_lock);
00212   LOG_END (" ");
00213 }
```

5.2.2.2 threadpool_create()

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

Definition at line 32 of file threadpool.c.

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

5.2.2.3 threadpool_create_default()

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

Definition at line 101 of file threadpool.c.

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

5.2.2.4 threadpool_destroy()

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

Definition at line 274 of file threadpool.c.

```
{
  LOG_INIT (" ");
  if (pool == NULL)
    {
      LOG_END (" ");
      return;
    }
  threadpool_stop (pool);

  pthread_mutex_lock (&pool->queue_lock);
  struct task *task = pool->task_queue_head;
  while (task != NULL)
    {
      struct task *tmp = task;
      task = task->next;
      free (tmp);
    }
  pthread_mutex_unlock (&pool->queue_lock);

  pthread_mutex_destroy (&pool->queue_lock);
  pthread_cond_destroy (&pool->queue_cond);
  pthread_cond_destroy (&pool->empty_cond);

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

5.2.2.5 threadpool_empty()

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

Definition at line 240 of file threadpool.c.

```
{
  LOG_INIT (" ");
  if (pool == NULL)
    {
      LOG_END (" ");
      return TRUE;
    }
  pthread_mutex_lock (&pool->queue_lock);
  int empty = (pool->task_queue_head == NULL);
  pthread_mutex_unlock (&pool->queue_lock);
  LOG_END (" ");
  return empty;
}
```

5.2.2.6 threadpool_init()

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

Definition at line 151 of file threadpool.c.

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

5.2.2.7 threadpool_stop()

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

Definition at line 216 of file threadpool.c.

```
00217 {
00218   LOG_INIT (" ");
00219   if (pool == NULL || !pool->started)
00220     {
00221       LOG_END (" ");
00222       return;
00223     }
00224   threadpool_wait_empty (pool);
00225 
00226   pthread_mutex_lock (&pool->queue_lock);
00227   pool->stop = TRUE;
00228   pthread_cond_broadcast (&pool->queue_cond);
00229   pthread_mutex_unlock (&pool->queue_lock);
00230 
00231   for (size_t i = 0; i < pool->num_threads; i++)
00232     {
00233       pthread_join (pool->threads[i], NULL);
00234     }
00235   pool->started = FALSE;
00236   LOG_END (" ");
00237 }
```

5.2.2.8 threadpool_wait_empty()

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

Definition at line 256 of file threadpool.c.

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

Chapter 6

Class Documentation

6.1 Node Struct Reference

Collaboration diagram for Node:

Public Attributes

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

6.1.1 Detailed Description

Definition at line 6 of file linked_list.c.

6.1.2 Member Data Documentation

6.1.2.1 children

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

Definition at line 10 of file linked_list.c.

6.1.2.2 next

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

Definition at line 11 of file linked_list.c.

6.1.2.3 ptr

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

Definition at line 8 of file linked_list.c.

6.1.2.4 size

```
size_t Node::size
```

Definition at line 9 of file linked_list.c.

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

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

6.2 request Struct Reference

Public Attributes

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

6.2.1 Detailed Description

Definition at line 32 of file low_saurion.c.

6.2.2 Member Data Documentation

6.2.2.1 client_socket

```
int request::client_socket
```

Definition at line 41 of file low_saurion.c.

6.2.2.2 event_type

```
int request::event_type
```

Definition at line 39 of file low_saurion.c.

6.2.2.3 iov

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

Definition at line 42 of file low_saurion.c.

6.2.2.4 iovec_count

```
size_t request::iovec_count
```

Definition at line 40 of file low_saurion.c.

6.2.2.5 next_iov

```
size_t request::next_iov
```

Definition at line 37 of file low_saurion.c.

6.2.2.6 next_offset

```
size_t request::next_offset
```

Definition at line 38 of file low_saurion.c.

6.2.2.7 prev

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

Definition at line 34 of file low_saurion.c.

6.2.2.8 prev_remain

```
size_t request::prev_remain
```

Definition at line 36 of file low_saurion.c.

6.2.2.9 prev_size

```
size_t request::prev_size
```

Definition at line 35 of file low_saurion.c.

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

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

6.3 saurion Struct Reference

Main structure for managing io_uring and socket events.

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

Collaboration diagram for saurion:

Classes

- struct saurion_callbacks

  Structure containing callback functions to handle socket events.

Public Attributes

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

6.3.1 Detailed Description

Main structure for managing io_uring and socket events.

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

Definition at line 148 of file low_saurion.h.

6.3.2 Member Data Documentation

6.3.2.1 efds

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

Eventfd descriptors used for internal signaling between threads.

Definition at line 157 of file low_saurion.h.

6.3.2.2 list

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

Linked list for storing active requests.

Definition at line 159 of file low_saurion.h.

6.3.2.3 m_rings

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

Array of mutexes to protect the io_uring rings.

Definition at line 153 of file low_saurion.h.

6.3.2.4 n_threads

```
uint32_t saurion::n_threads
```

Number of threads in the thread pool.

Definition at line 169 of file low_saurion.h.

6.3.2.5 next

```
uint32_t saurion::next
```

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

Definition at line 171 of file low_saurion.h.

6.3.2.6 pool

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

Thread pool for executing tasks in parallel.

Definition at line 167 of file low_saurion.h.

6.3.2.7 rings

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

Array of io_uring structures for managing the event queue.

Definition at line 151 of file low_saurion.h.

6.3.2.8 ss

```
int saurion::ss
```

Server socket descriptor for accepting connections.

Definition at line 155 of file low_saurion.h.

6.3.2.9 status

```
int saurion::status
```

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

Definition at line 165 of file low_saurion.h.

6.3.2.10 status_c

```
pthread_cond_t saurion::status_c
```

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

Definition at line 163 of file low_saurion.h.

6.3.2.11 status_m

```
pthread_mutex_t saurion::status_m
```

Mutex to protect the state of the structure.

Definition at line 161 of file low_saurion.h.

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

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

6.4 Saurion Class Reference

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

Collaboration diagram for Saurion:

Public Types

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

Public Member Functions

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

Private Attributes

- struct saurion ∗ s

6.4.1 Detailed Description

Definition at line 7 of file saurion.hpp.

6.4.2 Member Typedef Documentation

6.4.2.1 ClosedCb

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

Definition at line 14 of file saurion.hpp.

6.4.2.2 ConnectedCb

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

Definition at line 10 of file saurion.hpp.

6.4.2.3 ErrorCb

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

Definition at line 15 of file saurion.hpp.

6.4.2.4 ReadedCb

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

Definition at line 11 of file saurion.hpp.

6.4.2.5 WroteCb

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

Definition at line 13 of file saurion.hpp.

6.4.3 Constructor & Destructor Documentation

6.4.3.1 Saurion() [1/3]

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

Definition at line 5 of file saurion.cpp.

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

6.4.3.2 ∼Saurion()

```
Saurion::∼Saurion ( )
```

Definition at line 15 of file saurion.cpp.

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

6.4.3.3 Saurion() [2/3]

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

6.4.3.4 Saurion() [3/3]

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

6.4.4 Member Function Documentation

6.4.4.1 init()

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

Definition at line 18 of file saurion.cpp.

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

6.4.4.2 on_closed()

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

Definition at line 57 of file saurion.cpp.

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

6.4.4.3 on_connected()

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

Definition at line 33 of file saurion.cpp.

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

6.4.4.4 on_error()

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

Definition at line 65 of file saurion.cpp.

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

6.4.4.5 on_readed()

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

Definition at line 41 of file saurion.cpp.

```
{
  s->cb.on_readed = ncb;
  s->cb.on_readed_arg = arg;
  return this;
}
```

6.4.4.6 on_wrote()

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

Definition at line 49 of file saurion.cpp.

```
{
  s->cb.on_wrote = ncb;
  s->cb.on_wrote_arg = arg;
  return this;
}
```

6.4.4.7 operator=() [1/2]

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

6.4.4.8 operator=() [2/2]

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

6.4.4.9 send()

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

Definition at line 73 of file saurion.cpp.

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

6.4.4.10 stop()

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

Definition at line 27 of file saurion.cpp.

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

6.4.5 Member Data Documentation

6.4.5.1 s

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

Definition at line 38 of file saurion.hpp.

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

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

6.5 saurion::saurion_callbacks Struct Reference

Structure containing callback functions to handle socket events.

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

Public Attributes

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

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

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

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

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

    Callback for handling error events.
- void ∗ on_error_arg

6.5.1 Detailed Description

Structure containing callback functions to handle socket events.

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

Definition at line 181 of file low_saurion.h.

6.5.2 Member Data Documentation

6.5.2.1 on_closed

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

Callback for handling socket closures.

Parameters

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

Definition at line 221 of file low_saurion.h.

6.5.2.2 on_closed_arg

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

Additional argument for the close callback.

Definition at line 223 of file low_saurion.h.

6.5.2.3 on_connected

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

Callback for handling new connections.

Parameters

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

Definition at line 189 of file low_saurion.h.

6.5.2.4 on_connected_arg

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

Additional argument for the connection callback.

Definition at line 191 of file low_saurion.h.

6.5.2.5 on_error

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

Callback for handling error events.

Parameters

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

Definition at line 233 of file low_saurion.h.

6.5.2.6 on_error_arg

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

Additional argument for the error callback.

Definition at line 236 of file low_saurion.h.

6.5.2.7 on_readed

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

Callback for handling read events.

Parameters

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

Definition at line 201 of file low_saurion.h.

6.5.2.8 on_readed_arg

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

Additional argument for the read callback.

Definition at line 204 of file low_saurion.h.

6.5.2.9 on_wrote

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

Callback for handling write events.

Parameters

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

Definition at line 212 of file low_saurion.h.

6.5.2.10 on_wrote_arg

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

Additional argument for the write callback.

Definition at line 213 of file low_saurion.h.

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

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

6.6 saurion_callbacks Struct Reference

Structure containing callback functions to handle socket events.

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

Public Attributes

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

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

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

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

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

  Callback for handling error events.
- void ∗ on_error_arg

6.6.1 Detailed Description

Structure containing callback functions to handle socket events.

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

Definition at line 31 of file low_saurion.h.

6.6.2 Member Data Documentation

6.6.2.1 on_closed

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

Callback for handling socket closures.

Parameters

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

Definition at line 71 of file low_saurion.h.

6.6.2.2 on_closed_arg

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

Additional argument for the close callback.

Definition at line 73 of file low_saurion.h.

6.6.2.3 on_connected

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

Callback for handling new connections.

Parameters

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

Definition at line 39 of file low_saurion.h.

6.6.2.4 on_connected_arg

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

Additional argument for the connection callback.

Definition at line 41 of file low_saurion.h.

6.6.2.5 on_error

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

Callback for handling error events.

Parameters

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

Definition at line 83 of file low_saurion.h.

6.6.2.6 on_error_arg

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

Additional argument for the error callback.

Definition at line 86 of file low_saurion.h.

6.6.2.7 on_readed

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

Callback for handling read events.

Parameters

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

Definition at line 51 of file low_saurion.h.

6.6.2.8 on_readed_arg

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

Additional argument for the read callback.

Definition at line 54 of file low_saurion.h.

6.6.2.9 on_wrote

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

Callback for handling write events.

Parameters

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

Definition at line 62 of file low_saurion.h.

6.6.2.10 on_wrote_arg

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

Additional argument for the write callback.

Definition at line 63 of file low_saurion.h.

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

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

6.7 saurion_wrapper Struct Reference

Collaboration diagram for saurion_wrapper:

Public Attributes

- struct saurion ∗ s
- uint32_t sel

6.7.1 Detailed Description

Definition at line 51 of file low_saurion.c.

6.7.2 Member Data Documentation

6.7.2.1 s

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

Definition at line 53 of file low_saurion.c.

6.7.2.2 sel

```
uint32_t saurion_wrapper::sel
```

Definition at line 54 of file low_saurion.c.

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

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

6.8 task Struct Reference

Collaboration diagram for task:

Public Attributes

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

6.8.1 Detailed Description

Definition at line 11 of file threadpool.c.

6.8.2 Member Data Documentation

6.8.2.1 argument

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

Definition at line 14 of file threadpool.c.

6.8.2.2 function

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

Definition at line 13 of file threadpool.c.

6.8.2.3 next

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

Definition at line 15 of file threadpool.c.

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

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

6.9 threadpool Struct Reference

Collaboration diagram for threadpool:

Public Attributes

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

6.9.1 Detailed Description

Definition at line 18 of file threadpool.c.

6.9.2 Member Data Documentation

6.9.2.1 empty_cond

```
pthread_cond_t threadpool::empty_cond
```

Definition at line 26 of file threadpool.c.

6.9.2.2 num_threads

```
size_t threadpool::num_threads
```

Definition at line 21 of file threadpool.c.

6.9.2.3 queue_cond

```
pthread_cond_t threadpool::queue_cond
```

Definition at line 25 of file threadpool.c.

6.9.2.4 queue_lock

```
pthread_mutex_t threadpool::queue_lock
```

Definition at line 24 of file threadpool.c.

6.9.2.5 started

```
int threadpool::started
```

Definition at line 28 of file threadpool.c.

6.9.2.6 stop

```
int threadpool::stop
```

Definition at line 27 of file threadpool.c.

6.9.2.7 task_queue_head

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

Definition at line 22 of file threadpool.c.

6.9.2.8 task_queue_tail

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

Definition at line 23 of file threadpool.c.

6.9.2.9 threads

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

Definition at line 20 of file threadpool.c.

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

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

Chapter 7

File Documentation

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

`#include <stddef.h>`

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

Functions

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

7.1.1 Function Documentation

7.1.1.1 list_delete_node()

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

Definition at line 106 of file linked_list.c.

```
{
  pthread_mutex_lock (&list_mutex);
  struct Node *current = *head;
  struct Node *prev = NULL;

  if (current && current->ptr == ptr)
    {
      *head = current->next;
      free_node (current);
      pthread_mutex_unlock (&list_mutex);
      return;
    }

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

7.1.1.2 list_free()

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

Definition at line 138 of file linked_list.c.

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

7.1.1.3 list_insert()

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

Definition at line 65 of file linked_list.c.

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

7.2 linked_list.h

Go to the documentation of this file.

```
#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, const void *const ptr);

  void list_free (struct Node **head);

#ifdef __cplusplus
}
#endif

#endif // !LINKED_LIST_H
```

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

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

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

Classes

- struct saurion

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

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

  Structure containing callback functions to handle socket events.

Macros

- #define _POSIX_C_SOURCE 200809L
- #define PACKING_SZ 32

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

Functions

- int saurion_set_socket (int p)

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

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

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

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

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

  Sends a message through a socket using io_uring.

Variables

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

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

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

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

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

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

7.3.1 Variable Documentation

7.3.1.1 efds

```
int* efds
```

Eventfd descriptors used for internal signaling between threads.

Definition at line 7 of file low_saurion.h.

7.3.1.2 list

```
struct Node* list
```

Linked list for storing active requests.

Definition at line 9 of file low_saurion.h.

7.3.1.3 m_rings

```
pthread_mutex_t* m_rings
```

Array of mutexes to protect the io_uring rings.

Definition at line 3 of file low_saurion.h.

7.3.1.4 n_threads

```
uint32_t n_threads
```

Number of threads in the thread pool.

Definition at line 19 of file low_saurion.h.

7.3.1.5 next

```
uint32_t next
```

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

Definition at line 21 of file low_saurion.h.

7.3.1.6 on_closed

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

Callback for handling socket closures.

Parameters

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

Definition at line 38 of file low_saurion.h.

7.3.1.7 on_closed_arg

```
void * on_closed_arg
```

Additional argument for the close callback.

Definition at line 40 of file low_saurion.h.

7.3.1.8 on_connected

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

Callback for handling new connections.

Parameters

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

Definition at line 6 of file low_saurion.h.

7.3.1.9 on_connected_arg

```
void * on_connected_arg
```

Additional argument for the connection callback.

Definition at line 8 of file low_saurion.h.

7.3.1.10 on_error

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

Callback for handling error events.

Parameters

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

Definition at line 50 of file low_saurion.h.

7.3.1.11 on_error_arg

```
void * on_error_arg
```

Additional argument for the error callback.

Definition at line 53 of file low_saurion.h.

7.3.1.12 on_readed

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

Callback for handling read events.

Parameters

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

Definition at line 18 of file low_saurion.h.

7.3.1.13 on_readed_arg

```
void * on_readed_arg
```

Additional argument for the read callback.

Definition at line 21 of file low_saurion.h.

7.3.1.14 on_wrote

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

Callback for handling write events.

Parameters

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

Definition at line 29 of file low_saurion.h.

7.3.1.15 on_wrote_arg

```
void * on_wrote_arg
```

Additional argument for the write callback.

Definition at line 30 of file low_saurion.h.

7.3.1.16 pool

```
struct threadpool* pool
```

Thread pool for executing tasks in parallel.

Definition at line 17 of file low_saurion.h.

7.3.1.17 rings

```
struct io_uring* rings
```

Array of io_uring structures for managing the event queue.

Definition at line 1 of file low_saurion.h.

7.3.1.18 ss

```
int ss
```

Server socket descriptor for accepting connections.

Definition at line 5 of file low_saurion.h.

7.3.1.19 status

```
int status
```

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

Definition at line 15 of file low_saurion.h.

7.3.1.20 status_c

```
pthread_cond_t status_c
```

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

Definition at line 13 of file low_saurion.h.

7.3.1.21 status_m

```
pthread_mutex_t status_m
```

Mutex to protect the state of the structure.

Definition at line 11 of file low_saurion.h.

7.4 low_saurion.h

Go to the documentation of this file.

```

#ifndef LOW_SAURION_H
#define LOW_SAURION_H

#define _POSIX_C_SOURCE 200809L

#include <pthread.h>   // for pthread_mutex_t, pthread_cond_t
#include <stdint.h>    // for uint32_t
#include <sys/types.h> // for ssize_t

#ifdef __cplusplus
extern "C"
{
#endif

#define PACKING_SZ 32

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

    struct saurion_callbacks
    {
      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 saurion_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

```

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

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

`#include <stdint.h>`

Include dependency graph for low_saurion_secret.h:

Functions

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

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

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

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

7.6 low_saurion_secret.h

Go to the documentation of this file.

```
#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
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wpedantic"
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[];
};
#pragma GCC diagnostic pop
[[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
```

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

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

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

Classes

- class Saurion

7.8 saurion.hpp

Go to the documentation of this file.

```
#ifndef SAURION_HPP
#define SAURION_HPP

#include <stdint.h>    // for uint32_t
#include <sys/types.h> // for ssize_t

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 () const noexcept;

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

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

private:
  struct saurion *s;
};

#endif // !SAURION_HPP
```

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

#include <stddef.h>

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

Functions

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

7.10 threadpool.h

Go to the documentation of this file.

```

#ifndef THREADPOOL_H
#define THREADPOOL_H

#include <stddef.h> // for size_t

#ifdef __cplusplus
extern "C"
{
#endif

  struct threadpool;

  struct threadpool *threadpool_create (size_t num_threads);

  struct threadpool *threadpool_create_default (void);

  void threadpool_init (struct threadpool *pool);

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

  void threadpool_stop (struct threadpool *pool);

  int threadpool_empty (struct threadpool *pool);

  void threadpool_wait_empty (struct threadpool *pool);

  void threadpool_destroy (struct threadpool *pool);

#ifdef __cplusplus
}
#endif

#endif // !THREADPOOL_H

```

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

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

Include dependency graph for linked_list.c:

Classes

- struct Node

Functions

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

Variables

- pthread_mutex_t list_mutex = PTHREAD_MUTEX_INITIALIZER

7.11.1 Function Documentation

7.11.1.1 create_node()

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

Definition at line 17 of file linked_list.c.

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

7.11.1.2 free_node()

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

Definition at line 90 of file linked_list.c.

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

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

7.11.1.3 list_delete_node()

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

Definition at line 106 of file linked_list.c.

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

7.11.1.4 list_free()

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

Definition at line 138 of file linked_list.c.

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

7.11.1.5 list_insert()

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

Definition at line 65 of file linked_list.c.

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

7.11.2 Variable Documentation

7.11.2.1 list_mutex

```
pthread_mutex_t list_mutex = PTHREAD_MUTEX_INITIALIZER
```

Definition at line 14 of file linked_list.c.

7.12 linked_list.c

Go to the documentation of this file.

```
00001 #include "linked_list.h"
00002
00003 #include <pthread.h>
00004 #include <stdlib.h>
00005
00006 struct Node
00007 {
00008   void *ptr;
00009   size_t size;
00010   struct Node **children;
00011   struct Node *next;
00012 };
00013
00014 pthread_mutex_t list_mutex = PTHREAD_MUTEX_INITIALIZER;
00015
00016 struct Node *
00017 create_node (void *ptr, size_t amount, void **children)
00018 {
00019   struct Node *new_node = (struct Node *)malloc (sizeof (struct Node));
00020   if (!new_node)
00021     {
      return NULL;
    }
  new_node->ptr = ptr;
  new_node->size = amount;
  new_node->children = NULL;
  if (amount <= 0)
    {
      new_node->next = NULL;
      return new_node;
    }
  new_node->children
      = (struct Node **)malloc (sizeof (struct Node *) * amount);
  if (!new_node->children)
    {
      free (new_node);
      return NULL;
    }
  for (size_t i = 0; i < amount; ++i)
    {
      new_node->children[i] = (struct Node *)malloc (sizeof (struct Node));

      if (!new_node->children[i])
        {
          for (size_t j = 0; j < i; ++j)
            {
              free (new_node->children[j]);
            }
          free (new_node);
          return NULL;
        }
    }
  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);
  if (!new_node)
    {
      return 1;
    }
  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, const void *const ptr)
{
  pthread_mutex_lock (&list_mutex);
  struct Node *current = *head;
  struct Node *prev = NULL;

  if (current && current->ptr == ptr)
    {
      *head = current->next;
      free_node (current);
      pthread_mutex_unlock (&list_mutex);
      return;
    }

  while (current && current->ptr != ptr)
    {
      prev = current;
      current = current->next;
    }

  if (!current)
    {
      pthread_mutex_unlock (&list_mutex);
      return;
    }

  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);
}
```

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

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

Include dependency graph for low_saurion.c:

Classes

- struct request
- struct saurion_wrapper

Macros

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

Functions

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

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

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

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

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

  Creates an instance of the `saurion` structure.
- static int saurion_worker_master_loop_it (struct saurion *const s, struct sockaddr_in *client_addr, socklen_t *client_addr_len)
- void saurion_worker_master (void *arg)
- static int saurion_worker_slave_loop_it (struct saurion *const s, const int sel)
- void saurion_worker_slave (void *arg)
- int saurion_start (struct saurion *const s)

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

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

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

  Sends a message through a socket using io_uring.

Variables

- pthread_mutex_t print_mutex
- struct timespec TIMEOUT_RETRY_SPEC = { 0, TIMEOUT_RETRY ∗ 1000L }

7.13.1 Macro Definition Documentation

7.13.1.1 EV_ACC

```
#define EV_ACC 0
```

Definition at line 26 of file low_saurion.c.

7.13.1.2 EV_ERR

```
#define EV_ERR 4
```

Definition at line 30 of file low_saurion.c.

7.13.1.3 EV_REA

```
#define EV_REA 1
```

Definition at line 27 of file low_saurion.c.

7.13.1.4 EV_WAI

```
#define EV_WAI 3
```

Definition at line 29 of file low_saurion.c.

7.13.1.5 EV_WRI

```
#define EV_WRI 2
```

Definition at line 28 of file low_saurion.c.

7.13.1.6 MAX

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

Definition at line 46 of file low_saurion.c.

7.13.1.7 MIN

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

Definition at line 45 of file low_saurion.c.

7.13.2 Function Documentation

7.13.2.1 add_accept()

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

Definition at line 250 of file low_saurion.c.

```
{
  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]);
          nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
        }
      struct request *req = NULL;
      if (!set_request (&req, &s->list, 0, NULL, 0))
        {
          free (sqe);
          nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
          res = ERROR_CODE;
          continue;
        }
      req->client_socket = 0;
      req->event_type = EV_ACC;
      io_uring_prep_accept (sqe, s->ss, (struct sockaddr *const)ca, cal, 0);
      io_uring_sqe_set_data (sqe, req);
      if (io_uring_submit (&s->rings[0]) < 0)
        {
          free (sqe);
          list_delete_node (&s->list, req);
          nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
          res = ERROR_CODE;
          continue;
        }
      res = SUCCESS_CODE;
    }
  pthread_mutex_unlock (&s->m_rings[0]);
}
```

7.13.2.2 add_efd()

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

Definition at line 327 of file low_saurion.c.

```
00328 {
00329   add_fd (s, client_socket, sel);
00330 }
```

7.13.2.3 add_fd()

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

Definition at line 289 of file low_saurion.c.

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

7.13.2.4 add_read()

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

Definition at line 333 of file low_saurion.c.

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

7.13.2.5 add_read_continue()

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

Definition at line 340 of file low_saurion.c.

```
{
  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);
          nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
        }
      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]);
}
```

7.13.2.6 add_write()

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

Definition at line 376 of file low_saurion.c.

```
{
  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);
          nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
        }
      struct request *req = NULL;
      if (!set_request (&req, &s->list, strlen (str), (const void *const)str,
                        1))
        {
          free (sqe);
          res = ERROR_CODE;
          continue;
        }
      req->event_type = EV_WRI;
      req->client_socket = fd;
00400       io_uring_prep_writev (sqe, req->client_socket, req->iov,
00401                             req->iovec_count, 0);
00402       io_uring_sqe_set_data (sqe, req);
00403       if (io_uring_submit (ring) < 0)
00404         {
00405           free (sqe);
00406           list_delete_node (&s->list, req);
00407           res = ERROR_CODE;
00408           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00409           continue;
00410         }
00411       res = SUCCESS_CODE;
00412     }
00413   pthread_mutex_unlock (&s->m_rings[sel]);
00414 }
```

7.13.2.7 handle_accept()

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

Definition at line 418 of file low_saurion.c.

```
00419 {
00420   if (s->cb.on_connected)
00421     {
00422       s->cb.on_connected (fd, s->cb.on_connected_arg);
00423     }
00424 }
```

7.13.2.8 handle_close()

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

Definition at line 659 of file low_saurion.c.

```
00660 {
00661   if (s->cb.on_closed)
00662     {
00663       s->cb.on_closed (req->client_socket, s->cb.on_closed_arg);
00664     }
00665   close (req->client_socket);
00666 }
```

7.13.2.9 handle_error()

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

Definition at line 648 of file low_saurion.c.

```
00649 {
00650   if (s->cb.on_error)
00651     {
00652       const char *resp = "ERROR";
00653       s->cb.on_error (req->client_socket, resp, (ssize_t)strlen (resp),
00654                       s->cb.on_error_arg);
00655     }
00656 }
```

7.13.2.10 handle_read()

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

Definition at line 601 of file low_saurion.c.

```
00602 {
00603   void *msg = NULL;
00604   size_t len = 0;
00605   while (1)
00606     {
00607       if (!read_chunk (&msg, &len, req))
00608         {
00609           break;
00610         }
00611       if (req->next_iov || req->next_offset)
00612         {
00613           if (s->cb.on_readed && msg)
00614             {
00615               s->cb.on_readed (req->client_socket, msg, len,
00616                                 s->cb.on_readed_arg);
00617             }
00618           free (msg);
00619           msg = NULL;
00620           continue;
00621         }
00622       if (req->prev && req->prev_size && req->prev_remain)
00623         {
00624           add_read_continue (s, req, next (s));
00625           return;
00626         }
00627       if (s->cb.on_readed && msg)
00628         {
00629           s->cb.on_readed (req->client_socket, msg, len, s->cb.on_readed_arg);
00630         }
00631       free (msg);
00632       msg = NULL;
00633       break;
00634     }
00635   add_read (s, req->client_socket);
00636 }
```

7.13.2.11 handle_write()

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

Definition at line 639 of file low_saurion.c.

```
00640 {
00641   if (s->cb.on_wrote)
00642     {
00643       s->cb.on_wrote (fd, s->cb.on_wrote_arg);
00644     }
00645 }
```

7.13.2.12 htonll()

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

Definition at line 65 of file low_saurion.c.

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

```
  if (*(char *)&num == 42)
    {
      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;
}
```

7.13.2.13 next()

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

Definition at line 58 of file low_saurion.c.

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

7.13.2.14 ntohll()

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

Definition at line 78 of file low_saurion.c.

```
{
  int num = 42;
  if (*(char *)&num == 42)
    {
      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;
}
```

7.13.2.15 saurion_worker_master()

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

Definition at line 891 of file low_saurion.c.

```
{
  LOG_INIT (" ");
  struct saurion *const s = (struct saurion *)arg;
  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;
  pthread_cond_broadcast (&s->status_c);
  pthread_mutex_unlock (&s->status_m);
  while (1)
    {
      int ret
          = saurion_worker_master_loop_it (s, &client_addr, &client_addr_len);
      if (ret == ERROR_CODE || ret == CRITICAL_CODE)
        {
          break;
        }
    }
  pthread_mutex_lock (&s->status_m);
  --s->status;
  pthread_cond_signal (&s->status_c);
  pthread_mutex_unlock (&s->status_m);
  LOG_END (" ");
  return;
}
```

7.13.2.16 saurion_worker_master_loop_it()

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

Definition at line 823 of file low_saurion.c.

```
{
  LOG_INIT (" ");
  struct io_uring ring = s->rings[0];
  struct io_uring_cqe *cqe = NULL;
  int ret = io_uring_wait_cqe (&ring, &cqe);
  if (ret < 0)
    {
      free (cqe);
      LOG_END (" ");
      return CRITICAL_CODE;
    }
  struct request *req = (struct request *)cqe->user_data;
  if (!req)
    {
      io_uring_cqe_seen (&s->rings[0], cqe);
      LOG_END (" ");
      return SUCCESS_CODE;
    }
  if (cqe->res < 0)
    {
      list_delete_node (&s->list, req);
      LOG_END (" ");
      return CRITICAL_CODE;
    }
  if (req->client_socket == s->efds[0])
    {
      io_uring_cqe_seen (&s->rings[0], cqe);
      list_delete_node (&s->list, req);
      LOG_END (" ");
      return ERROR_CODE;
    }
  io_uring_cqe_seen (&s->rings[0], cqe);
  switch (req->event_type)
    {
    case EV_ACC:
      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 EV_REA:
      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 EV_WRI:
```

```
00882       handle_write (s, req->client_socket);
00883       list_delete_node (&s->list, req);
00884       break;
00885     }
00886   LOG_END (" ");
00887   return SUCCESS_CODE;
00888 }
```

7.13.2.17 saurion_worker_slave()

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

Definition at line 986 of file low_saurion.c.

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

7.13.2.18 saurion_worker_slave_loop_it()

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

Definition at line 924 of file low_saurion.c.

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

```
    }
  if (cqe->res < 0)
    {
      list_delete_node (&s->list, req);
      LOG_END (" ");
      return CRITICAL_CODE;
    }
  if (req->client_socket == s->efds[sel])
    {
      io_uring_cqe_seen (&ring, cqe);
      list_delete_node (&s->list, req);
      LOG_END (" ");
      return ERROR_CODE;
    }
  io_uring_cqe_seen (&ring, cqe);
  switch (req->event_type)
    {
    case EV_REA:
      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 EV_WRI:
      handle_write (s, req->client_socket);
      list_delete_node (&s->list, req);
      break;
    }
  LOG_END (" ");
  return SUCCESS_CODE;
}
```

7.13.3 Variable Documentation

7.13.3.1 print_mutex

```
pthread_mutex_t print_mutex
```

Definition at line 47 of file low_saurion.c.

7.13.3.2 TIMEOUT_RETRY_SPEC

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

Definition at line 49 of file low_saurion.c.

7.14 low_saurion.c

Go to the documentation of this file.

```
#include "low_saurion.h"
#include "config.h"      // for ERROR_CODE, SUCCESS_CODE, CHUNK_SZ
#include "linked_list.h" // for list_delete_node, list_free, list_insert
#include "threadpool.h"  // for threadpool_add, threadpool_create

#include <arpa/inet.h>             // for htonl, ntohl, htons
#include <bits/socket-constants.h> // for SOL_SOCKET, SO_REUSEADDR
#include <liburing.h>          // for io_uring_get_sqe, io_uring, io_uring_...
#include <liburing/io_uring.h> // for io_uring_cqe
#include <nanologger.h>        // for LOG_END, LOG_INIT
#include <netinet/in.h>        // for sockaddr_in, INADDR_ANY, in_addr
#include <pthread.h>           // for pthread_mutex_lock, pthread_mutex_unlock
#include <stdint.h>            // for uint32_t, uint64_t, uint8_t
#include <stdio.h>             // for NULL
#include <stdlib.h>            // for free, malloc
#include <string.h>            // for memset, memcpy, strlen
#include <sys/eventfd.h>       // for eventfd, EFD_NONBLOCK
#include <sys/socket.h>        // for socklen_t, bind, listen, setsockopt
#include <sys/uio.h>           // for iovec
#include <time.h>              // for nanosleep
#include <unistd.h>            // for close, write

struct Node;
struct iovec;

#define EV_ACC 0
#define EV_REA 1
#define EV_WRI 2
#define EV_WAI 3
#define EV_ERR 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 timespec TIMEOUT_RETRY_SPEC = { 0, TIMEOUT_RETRY * 1000L };

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)
    {
      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)
    {
      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) + sizeof (uint8_t));
    }
  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, 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]);
          nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
        }
      struct request *req = NULL;
      if (!set_request (&req, &s->list, 0, NULL, 0))
        {
          free (sqe);
          nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
          res = ERROR_CODE;
          continue;
        }
      req->client_socket = 0;
      req->event_type = EV_ACC;
      io_uring_prep_accept (sqe, s->ss, (struct sockaddr *const)ca, cal, 0);
      io_uring_sqe_set_data (sqe, req);
      if (io_uring_submit (&s->rings[0]) < 0)
        {
          free (sqe);
          list_delete_node (&s->list, req);
          nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
          res = ERROR_CODE;
          continue;
        }
      res = SUCCESS_CODE;
    }
  pthread_mutex_unlock (&s->m_rings[0]);
}

static void
add_fd (struct saurion *const s, int client_socket, 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);
          nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
        }
      struct request *req = NULL;
      if (!set_request (&req, &s->list, CHUNK_SZ, NULL, 0))
        {
          free (sqe);
          res = ERROR_CODE;
          continue;
        }
      req->event_type = EV_REA;
      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_efd (struct saurion *const s, const int client_socket, int sel)
{
  add_fd (s, client_socket, sel);
}

static void
add_read (struct saurion *const s, const int client_socket)
{
  int sel = next (s);
  add_fd (s, client_socket, 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);
          nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
        }
      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);
          nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
        }
      struct request *req = NULL;
      if (!set_request (&req, &s->list, strlen (str), (const void *const)str,
                        1))
        {
          free (sqe);
          res = ERROR_CODE;
          continue;
        }
      req->event_type = EV_WRI;
      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;
          nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
          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)
{
  if (req->iovec_count == 0)
```

```
    {
      return ERROR_CODE;
    }

  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;
  size_t cont_rem = 0;
  size_t curr_iov = 0;
  size_t curr_iov_off = 0;
  size_t dest_off = 0;
  void *dest_ptr = NULL;
  if (req->prev && req->prev_size && req->prev_remain)
    {
      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)
    {
      curr_iov = req->next_iov;
      curr_iov_off = req->next_offset;
      cont_sz = *(
          (size_t *)(((uint8_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
    {
      curr_iov = 0;
      curr_iov_off = 0;
      cont_sz = *(
          (size_t *)(((uint8_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;

  uint8_t ok = 1UL;
  while (1)
    {
      curr_iov_msg_rem
```

```
          = MIN (cont_rem, (req->iov[curr_iov].iov_len - curr_iov_off));
      memcpy ((uint8_t *)dest_ptr + dest_off,
              ((uint8_t *)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)
        {
          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;
        }
    }

  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 *)(((uint8_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;
        }
    }

  if (ok)
    {
      return SUCCESS_CODE;
    }
  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;
}

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;
        }
      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;
        }
      if (req->prev && req->prev_size && req->prev_remain)
        {
          add_read_continue (s, req, next (s));
          return;
        }
      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
saurion_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;
    }

  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);

  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)
{
  LOG_INIT (" ");
  struct saurion *p = (struct saurion *)malloc (sizeof (struct saurion));
  if (!p)
    {
      LOG_END (" ");
      return NULL;
    }
  int ret = 0;
  ret = pthread_mutex_init (&p->status_m, NULL);
  if (ret)
    {
      free (p);
      LOG_END (" ");
      return NULL;
    }
  ret = pthread_cond_init (&p->status_c, NULL);
  if (ret)
    {
      free (p);
      LOG_END (" ");
      return NULL;
    }
  p->m_rings
      = (pthread_mutex_t *)malloc (n_threads * sizeof (pthread_mutex_t));
  if (!p->m_rings)
    {
      free (p);
      LOG_END (" ");
      return NULL;
    }
  for (uint32_t i = 0; i < n_threads; ++i)
    {
      pthread_mutex_init (&(p->m_rings[i]), NULL);
    }
  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;
  p->efds = (int *)malloc (sizeof (int) * p->n_threads);
  if (!p->efds)
    {
      free (p->m_rings);
      free (p);
      LOG_END (" ");
      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);
          LOG_END (" ");
          return NULL;
        }
    }
  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);
      LOG_END (" ");
      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);
          LOG_END (" ");
          return NULL;
        }
    }
  p->pool = threadpool_create (p->n_threads);
  LOG_END (" ");
  return p;
}

[[nodiscard]]
static int
saurion_worker_master_loop_it (struct saurion *const s,
                               struct sockaddr_in *client_addr,
                               socklen_t *client_addr_len)
{
  LOG_INIT (" ");
  struct io_uring ring = s->rings[0];
  struct io_uring_cqe *cqe = NULL;
  int ret = io_uring_wait_cqe (&ring, &cqe);
  if (ret < 0)
    {
      free (cqe);
      LOG_END (" ");
      return CRITICAL_CODE;
    }
  struct request *req = (struct request *)cqe->user_data;
  if (!req)
    {
      io_uring_cqe_seen (&s->rings[0], cqe);
      LOG_END (" ");
      return SUCCESS_CODE;
    }
  if (cqe->res < 0)
    {
      list_delete_node (&s->list, req);
      LOG_END (" ");
      return CRITICAL_CODE;
    }
  if (req->client_socket == s->efds[0])
    {
      io_uring_cqe_seen (&s->rings[0], cqe);
      list_delete_node (&s->list, req);
      LOG_END (" ");
      return ERROR_CODE;
    }
  io_uring_cqe_seen (&s->rings[0], cqe);
  switch (req->event_type)
    {
    case EV_ACC:
      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 EV_REA:
      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 EV_WRI:
      handle_write (s, req->client_socket);
      list_delete_node (&s->list, req);
      break;
    }
  LOG_END (" ");
  return SUCCESS_CODE;
}

void
saurion_worker_master (void *arg)
{
  LOG_INIT (" ");
  struct saurion *const s = (struct saurion *)arg;
  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;
  pthread_cond_broadcast (&s->status_c);
  pthread_mutex_unlock (&s->status_m);
  while (1)
    {
      int ret
          = saurion_worker_master_loop_it (s, &client_addr, &client_addr_len);
      if (ret == ERROR_CODE || ret == CRITICAL_CODE)
        {
          break;
        }
    }
  pthread_mutex_lock (&s->status_m);
  --s->status;
  pthread_cond_signal (&s->status_c);
  pthread_mutex_unlock (&s->status_m);
  LOG_END (" ");
  return;
}

[[nodiscard]]
static int
saurion_worker_slave_loop_it (struct saurion *const s, const int sel)
{
  LOG_INIT (" ");
  struct io_uring ring = s->rings[sel];
  struct io_uring_cqe *cqe = NULL;

  add_efd (s, s->efds[sel], sel);
  int ret = io_uring_wait_cqe (&ring, &cqe);
  if (ret < 0)
    {
      free (cqe);
      LOG_END (" ");
      return CRITICAL_CODE;
    }
  struct request *req = (struct request *)cqe->user_data;
  if (!req)
    {
      io_uring_cqe_seen (&ring, cqe);
      LOG_END (" ");
      return SUCCESS_CODE;
    }
  if (cqe->res < 0)
    {
      list_delete_node (&s->list, req);
      LOG_END (" ");
      return CRITICAL_CODE;
    }
  if (req->client_socket == s->efds[sel])
    {
      io_uring_cqe_seen (&ring, cqe);
      list_delete_node (&s->list, req);
      LOG_END (" ");
      return ERROR_CODE;
    }
  io_uring_cqe_seen (&ring, cqe);
  switch (req->event_type)
    {
    case EV_REA:
      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 EV_WRI:
      handle_write (s, req->client_socket);
      list_delete_node (&s->list, req);
      break;
    }
  LOG_END (" ");
  return SUCCESS_CODE;
}

void
saurion_worker_slave (void *arg)
{
  LOG_INIT (" ");
  struct saurion_wrapper *const ss = (struct saurion_wrapper *)arg;
  struct saurion *s = ss->s;
  const int sel = ss->sel;
  free (ss);

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

  pthread_mutex_lock (&s->status_m);
  ++s->status;
  pthread_cond_broadcast (&s->status_c);
  pthread_mutex_unlock (&s->status_m);
  while (1)
    {
      int res = saurion_worker_slave_loop_it (s, sel);
      if (res == ERROR_CODE || res == CRITICAL_CODE)
        {
          break;
        }
    }
  pthread_mutex_lock (&s->status_m);
  --s->status;
  pthread_cond_signal (&s->status_c);
  pthread_mutex_unlock (&s->status_m);
  LOG_END (" ");
  return;
}

[[nodiscard]]
int
saurion_start (struct saurion *const s)
{
  pthread_mutex_init (&print_mutex, NULL);
  threadpool_init (s->pool);
  threadpool_add (s->pool, saurion_worker_master, s);
  struct saurion_wrapper *ss = NULL;
  for (uint32_t i = 1; i < s->n_threads; ++i)
    {
      ss = (struct saurion_wrapper *)malloc (sizeof (struct saurion_wrapper));
      if (!ss)
        {
          return ERROR_CODE;
        }
      ss->s = s;
      ss->sel = i;
      threadpool_add (s->pool, saurion_worker_slave, ss);
    }
  pthread_mutex_lock (&s->status_m);
  while (s->status < (int)s->n_threads)
    {
      pthread_cond_wait (&s->status_c, &s->status_m);
    }
  pthread_mutex_unlock (&s->status_m);
  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)
        {
          nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
        }
    }
  threadpool_wait_empty (s->pool);
}

void
saurion_destroy (struct saurion *const s)
{
  pthread_mutex_lock (&s->status_m);
  while (s->status > 0)
    {
      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));
}
```

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

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

Include dependency graph for main.c:

7.16 main.c

Go to the documentation of this file.

```
#include <pthread.h> // for pthread_create, pthread_join, pthread_t
#include <stdio.h>   // for printf, fprintf, NULL, stderr

int counter = 0;

void *
increment (void *arg)
{
  int id = *((int *)arg);
  for (int i = 0; i < 100000; ++i)
    {
      counter++;
      if (i % 10000 == 0)
        {
          printf ("Thread %d at iteration %d\n", id, i);
        }
    }
  printf ("Thread %d finished\n", id);
  return NULL;
}

int
main ()
{
  pthread_t t1;
  pthread_t t2;
  int id1 = 1;
  int id2 = 2;

  printf ("Starting threads...\n");

  if (pthread_create (&t1, NULL, increment, &id1))
    {
      fprintf (stderr, "Error creating thread 1\n");
      return 1;
    }
  if (pthread_create (&t2, NULL, increment, &id2))
    {
      fprintf (stderr, "Error creating thread 2\n");
      return 1;
    }

  printf ("Waiting for thread 1 to join...\n");
  if (pthread_join (t1, NULL))
    {
      fprintf (stderr, "Error joining thread 1\n");
      return 2;
    }
  printf ("Thread 1 joined\n");

  printf ("Waiting for thread 2 to join...\n");
  if (pthread_join (t2, NULL))
    {
      fprintf (stderr, "Error joining thread 2\n");
      return 2;
    }
  printf ("Thread 2 joined\n");

  printf ("Final counter value:  %d\n", counter);
  return 0;
}
```

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

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

Include dependency graph for saurion.cpp:

7.18 saurion.cpp

Go to the documentation of this file.

```
#include "saurion.hpp"

#include "low_saurion.h" // for saurion, saurion_create, saurion_destroy

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 () const 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);
}
```

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

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

Include dependency graph for threadpool.c:

Classes

- struct task
- struct threadpool

Macros

- #define TRUE 1
- #define FALSE 0

Functions

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

7.19.1 Macro Definition Documentation

7.19.1.1 FALSE

```
#define FALSE 0
```

Definition at line 9 of file threadpool.c.

7.19.1.2 TRUE

```
#define TRUE 1
```

Definition at line 8 of file threadpool.c.

7.19.2 Function Documentation

7.19.2.1 threadpool_worker()

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

Definition at line 107 of file threadpool.c.

```
{
  LOG_INIT (" ");
  struct threadpool *pool = (struct threadpool *)arg;
  while (TRUE)
    {
      pthread_mutex_lock (&pool->queue_lock);
      while (pool->task_queue_head == NULL && !pool->stop)
        {
          pthread_cond_wait (&pool->queue_cond, &pool->queue_lock);
        }

      if (pool->stop && pool->task_queue_head == NULL)
        {
          pthread_mutex_unlock (&pool->queue_lock);
          break;
        }

      struct task *task = pool->task_queue_head;
      if (task != NULL)
        {
          pool->task_queue_head = task->next;
          if (pool->task_queue_head == NULL)
            pool->task_queue_tail = NULL;

          if (pool->task_queue_head == NULL)
            {
              pthread_cond_signal (&pool->empty_cond);
            }
        }
      pthread_mutex_unlock (&pool->queue_lock);

      if (task != NULL)
        {
          task->function (task->argument);
          free (task);
        }
    }
  LOG_END (" ");
  pthread_exit (NULL);
  return NULL;
}
```

7.20 threadpool.c

Go to the documentation of this file.

```
#include "threadpool.h"
#include "config.h"     // for NUM_CORES
#include <nanologger.h> // for LOG_END, LOG_INIT
#include <pthread.h>    // for pthread_mutex_unlock, pthread_mutex_lock
#include <stdio.h>      // for perror
#include <stdlib.h>     // for free, malloc

#define TRUE 1
#define FALSE 0

struct task
{
  void (*function) (void *);
  void *argument;
  struct task *next;
};

struct threadpool
{
  pthread_t *threads;
  size_t num_threads;
  struct task *task_queue_head;
  struct task *task_queue_tail;
  pthread_mutex_t queue_lock;
  pthread_cond_t queue_cond;
  pthread_cond_t empty_cond;
  int stop;
```

```c
  int started;
};

struct threadpool *
threadpool_create (size_t num_threads)
{
  LOG_INIT (" ");
  struct threadpool *pool = malloc (sizeof (struct threadpool));
  if (pool == NULL)
    {
      perror ("Failed to allocate threadpool");
      LOG_END (" ");
      return NULL;
    }
  if (num_threads < 3)
    {
      num_threads = 3;
    }
  if (num_threads > NUM_CORES)
    {
      num_threads = NUM_CORES;
    }

  pool->num_threads = num_threads;
  pool->threads = malloc (sizeof (pthread_t) * num_threads);
  if (pool->threads == NULL)
    {
      perror ("Failed to allocate threads array");
      free (pool);
      LOG_END (" ");
      return NULL;
    }

  pool->task_queue_head = NULL;
  pool->task_queue_tail = NULL;
  pool->stop = FALSE;
  pool->started = FALSE;

  if (pthread_mutex_init (&pool->queue_lock, NULL) != 0)
    {
      perror ("Failed to initialize mutex");
      free (pool->threads);
      free (pool);
      LOG_END (" ");
      return NULL;
    }

  if (pthread_cond_init (&pool->queue_cond, NULL) != 0)
    {
      perror ("Failed to initialize condition variable");
      pthread_mutex_destroy (&pool->queue_lock);
      free (pool->threads);
      free (pool);
      LOG_END (" ");
      return NULL;
    }

  if (pthread_cond_init (&pool->empty_cond, NULL) != 0)
    {
      perror ("Failed to initialize empty condition variable");
      pthread_mutex_destroy (&pool->queue_lock);
      pthread_cond_destroy (&pool->queue_cond);
      free (pool->threads);
      free (pool);
      LOG_END (" ");
      return NULL;
    }

  LOG_END (" ");
  return pool;
}

struct threadpool *
threadpool_create_default (void)
{
  return threadpool_create (NUM_CORES);
}

void *
threadpool_worker (void *arg)
{
  LOG_INIT (" ");
  struct threadpool *pool = (struct threadpool *)arg;
  while (TRUE)
    {
      pthread_mutex_lock (&pool->queue_lock);
      while (pool->task_queue_head == NULL && !pool->stop)
```

```
        {
          pthread_cond_wait (&pool->queue_cond, &pool->queue_lock);
        }

      if (pool->stop && pool->task_queue_head == NULL)
        {
          pthread_mutex_unlock (&pool->queue_lock);
          break;
        }

      struct task *task = pool->task_queue_head;
      if (task != NULL)
        {
          pool->task_queue_head = task->next;
          if (pool->task_queue_head == NULL)
            pool->task_queue_tail = NULL;

          if (pool->task_queue_head == NULL)
            {
              pthread_cond_signal (&pool->empty_cond);
            }
        }
      pthread_mutex_unlock (&pool->queue_lock);

      if (task != NULL)
        {
          task->function (task->argument);
          free (task);
        }
    }
  LOG_END (" ");
  pthread_exit (NULL);
  return NULL;
}

void
threadpool_init (struct threadpool *pool)
{
  LOG_INIT (" ");
  if (pool == NULL || pool->started)
    {
      LOG_END (" ");
      return;
    }
  for (size_t i = 0; i < pool->num_threads; i++)
    {
      if (pthread_create (&pool->threads[i], NULL, threadpool_worker,
                          (void *)pool)
          != 0)
        {
          perror ("Failed to create thread");
          pool->stop = TRUE;
          break;
        }
    }
  pool->started = TRUE;
  LOG_END (" ");
}

void
threadpool_add (struct threadpool *pool, void (*function) (void *),
                void *argument)
{
  LOG_INIT (" ");
  if (pool == NULL || function == NULL)
    {
      LOG_END (" ");
      return;
    }

  struct task *new_task = malloc (sizeof (struct task));
  if (new_task == NULL)
    {
      perror ("Failed to allocate task");
      LOG_END (" ");
      return;
    }

  new_task->function = function;
  new_task->argument = argument;
  new_task->next = NULL;

  pthread_mutex_lock (&pool->queue_lock);

  if (pool->task_queue_head == NULL)
    {
      pool->task_queue_head = new_task;
      pool->task_queue_tail = new_task;
    }
  else
    {
      pool->task_queue_tail->next = new_task;
      pool->task_queue_tail = new_task;
    }
  pthread_cond_signal (&pool->queue_cond);

  pthread_mutex_unlock (&pool->queue_lock);
  LOG_END (" ");
}

void
threadpool_stop (struct threadpool *pool)
{
  LOG_INIT (" ");
  if (pool == NULL || !pool->started)
    {
      LOG_END (" ");
      return;
    }
  threadpool_wait_empty (pool);

  pthread_mutex_lock (&pool->queue_lock);
  pool->stop = TRUE;
  pthread_cond_broadcast (&pool->queue_cond);
  pthread_mutex_unlock (&pool->queue_lock);

  for (size_t i = 0; i < pool->num_threads; i++)
    {
      pthread_join (pool->threads[i], NULL);
    }
  pool->started = FALSE;
  LOG_END (" ");
}

int
threadpool_empty (struct threadpool *pool)
{
  LOG_INIT (" ");
  if (pool == NULL)
    {
      LOG_END (" ");
      return TRUE;
    }
  pthread_mutex_lock (&pool->queue_lock);
  int empty = (pool->task_queue_head == NULL);
  pthread_mutex_unlock (&pool->queue_lock);
  LOG_END (" ");
  return empty;
}

void
threadpool_wait_empty (struct threadpool *pool)
{
  LOG_INIT (" ");
  if (pool == NULL)
    {
      LOG_END (" ");
      return;
    }
  pthread_mutex_lock (&pool->queue_lock);
  while (pool->task_queue_head != NULL)
    {
      pthread_cond_wait (&pool->empty_cond, &pool->queue_lock);
    }
  pthread_mutex_unlock (&pool->queue_lock);
  LOG_END (" ");
}

void
threadpool_destroy (struct threadpool *pool)
{
  LOG_INIT (" ");
  if (pool == NULL)
    {
      LOG_END (" ");
      return;
    }
  threadpool_stop (pool);

  pthread_mutex_lock (&pool->queue_lock);
  struct task *task = pool->task_queue_head;
  while (task != NULL)
    {
      struct task *tmp = task;
      task = task->next;
      free (tmp);
    }
  pthread_mutex_unlock (&pool->queue_lock);

  pthread_mutex_destroy (&pool->queue_lock);
  pthread_cond_destroy (&pool->queue_cond);
  pthread_cond_destroy (&pool->empty_cond);

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

Index