Saurion

Generated by Doxygen 1.9.4

Chapter 1

Module Index

1.1 Modules

Here is a list of all modules:

Chapter 2

Class Index

2.1 Class List

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

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Module Documentation

4.1 LinkedList

A module for managing a thread-safe linked list.

Classes

- struct Node

  Represents a node in the linked list.

Functions

- int list_insert (struct Node **head, void *ptr, const uint64_t amount, void **children)

  Inserts a new node into the linked list.
- void list_delete_node (struct Node **head, const void *const ptr)

  Deletes a node from the linked list.
- void list_free (struct Node **head)

  Frees the entire linked list.

4.1.1 Detailed Description

A module for managing a thread-safe linked list.

This module provides functions to create, insert, delete, and free nodes in a linked list. It is thread-safe, using a mutex to ensure proper synchronization.

4.1.1.0.1 General Diagram of the Linked List:

```
Head -> [Node(ptr=A, size=2)] -> [Node(ptr=B, size=0)] -> NULL
  |                      |
  +-> children[0] -> [Child Node(ptr=C)] -> NULL
  +-> children[1] -> [Child Node(ptr=D)] -> NULL
```

4.1.1.0.2 Example Usage: #include "linked_list.h"

```
#include <stdio.h>
int main() {
    struct Node *head = NULL;
    int data1 = 42, data2 = 24;
    void *children[] = {&data1, &data2};
    if (list_insert(&head, (void *)&data1, 2, children) == SUCCESS_CODE) {
        printf("Node inserted successfully!\n");
    } else {
        printf("Error inserting node.\n");
    }
    list_delete_node(&head, (void *)&data1);
    list_free(&head);
    return 0;
}
```

Author

Israel

Date

2024

4.1.2 Function Documentation

4.1.2.1 list_delete_node()

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

Deletes a node from the linked list.

This function removes the first node containing the specified data and frees its memory.

Parameters

head	Pointer to the head of the linked list.
ptr	Pointer to the data to identify the node to delete.

4.1.2.1.1 Delete Operation Diagram: Before deletion:

```
Head -> [Node(ptr=A, size=2)] -> [Node(ptr=B, size=0)] -> NULL
        |
        +-> children[0] -> [Child Node(ptr=C)] -> NULL
        +-> children[1] -> [Child Node(ptr=D)] -> NULL
After deleting node with ptr=A:
Head -> [Node(ptr=B, size=0)] -> NULL
```

Definition at line 114 of file linked_list.c.

```
00115 {
00116   pthread_mutex_lock (&list_mutex);
00117   struct Node *current = *head;
00118   struct Node *prev = NULL;
00119
00120   if (current && current->ptr == ptr)
00121     {
00122       *head = current->next;
00123       free_node (current);
00124       pthread_mutex_unlock (&list_mutex);
```

```
00125       return;
00126     }
00127
00128   while (current && current->ptr != ptr)
00129     {
00130       prev = current;
00131       current = current->next;
00132     }
00133
00134   if (!current)
00135     {
00136       pthread_mutex_unlock (&list_mutex);
00137       return;
00138     }
00139
00140   prev->next = current->next;
00141   free_node (current);
00142   pthread_mutex_unlock (&list_mutex);
00143 }
```

4.1.2.2 list_free()

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

Frees the entire linked list.

This function traverses the linked list, freeing each node and its associated resources.

Parameters

head	Pointer to the head of the linked list.

4.1.2.2.1 Free Operation Diagram: Before freeing:

```
Head -> [Node(ptr=A, size=2)] -> [Node(ptr=B, size=0)] -> NULL
        |
        +-> children[0] -> [Child Node(ptr=C)] -> NULL
        +-> children[1] -> [Child Node(ptr=D)] -> NULL
After freeing:
Head -> NULL
```

Definition at line 147 of file linked_list.c.

```
00148 {
00149   pthread_mutex_lock (&list_mutex);
00150   struct Node *current = *head;
00151   struct Node *next;
00152
00153   while (current)
00154     {
00155       next = current->next;
00156       free_node (current);
00157       current = next;
00158     }
00159
00160   *head = NULL;
00161   pthread_mutex_unlock (&list_mutex);
00162 }
```

4.1.2.3 list_insert()

```
int list_insert (
              struct Node ** head,
```

```
            void * ptr,
            const uint64_t amount,
            void ** children )
```

Inserts a new node into the linked list.

This function creates and inserts a new node with the given data into the list.

Parameters

head	Pointer to the head of the linked list.
ptr	Pointer to the data to be stored in the new node.
amount	Number of children nodes to allocate.
children	Array of pointers to data for the children nodes.

Returns

`SUCCESS_CODE` on success, or `ERROR_CODE` on failure.

4.1.2.3.1 Insert Operation Diagram: `Initial state:`

```
Head -> NULL
After inserting a node:
Head -> [Node(ptr=A, size=2)] -> NULL
        |
        +-> children[0] -> [Child Node(ptr=C)] -> NULL
        +-> children[1] -> [Child Node(ptr=D)] -> NULL
```

Definition at line 70 of file linked_list.c.

```
00072 {
00073   struct Node *new_node = create_node (ptr, amount, children);
00074   if (!new_node)
00075     {
00076       return ERROR_CODE;
00077     }
00078   pthread_mutex_lock (&list_mutex);
00079   if (!*head)
00080     {
00081       *head = new_node;
00082       pthread_mutex_unlock (&list_mutex);
00083       return SUCCESS_CODE;
00084     }
00085   struct Node *temp = *head;
00086   while (temp->next)
00087     {
00088       temp = temp->next;
00089     }
00090   temp->next = new_node;
00091   pthread_mutex_unlock (&list_mutex);
00092   return SUCCESS_CODE;
00093 }
```

4.2 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_callbacks

  Structure containing callback functions to handle socket events.
- 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 (const 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, const uint64_t amount, const uint64_t pos, const uint64_t size, void **chd_ptr)
- int initialize_iovec (struct iovec *iov, const uint64_t amount, const uint64_t pos, const void *msg, const uint64_t size, const uint8_t h)

  Initializes a specified `iovec` structure with a message fragment.
- int set_request (struct request **r, struct Node **l, uint64_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, uint64_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, uint64_t amount)

4.2.1 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.

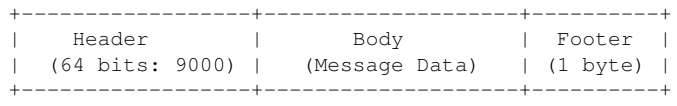
When these messages are sent to the kernel, they are divided into chunks using `iovec`. Each chunk can hold a maximum of 8192 bytes and contains two fields:

- `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.

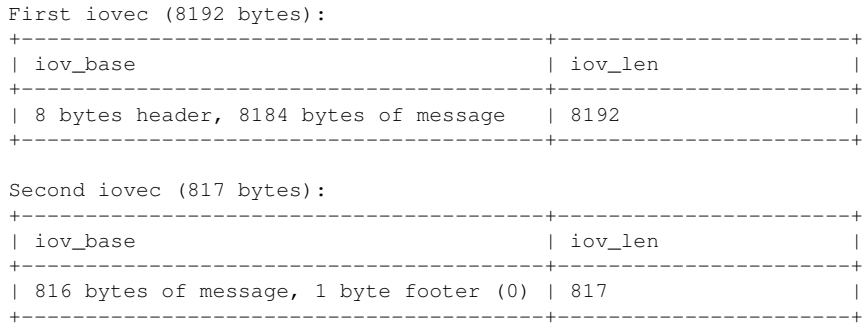
For the message with 9000 bytes, the `iovec` division would look like this:

- 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:



The structure of the `iovec` division is:



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

Basic usage example:

```
struct saurion *s = saurion_create(4);
if (saurion_start(s) != 0) {
    handle_error();
}
saurion_send(s, socket_fd, "Hello, World!");
saurion_stop(s);
saurion_destroy(s);
```

1. Create the saurion structure with 4 threads
2. Start event processing
3. Send a message through a socket
4. Stop event processing
5. Destroy the structure and free resources

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.

4.2.2 Macro Definition Documentation

4.2.2.1 _POSIX_C_SOURCE

```
#define _POSIX_C_SOURCE 200809L
```

Definition at line 108 of file low_saurion.h.

4.2.2.2 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 141 of file low_saurion.h.

4.2.3 Function Documentation

4.2.3.1 allocate_iovec()

```
int allocate_iovec (
            struct iovec * iov,
            const uint64_t amount,
            const uint64_t pos,
            const uint64_t size,
            void ** chd_ptr )
```

Definition at line 154 of file low_saurion.c.

```
00156 {
00157   if (!iov || !chd_ptr)
00158     {
00159       return ERROR_CODE;
00160     }
00161   iov->iov_base = malloc (CHUNK_SZ);
00162   if (!iov->iov_base)
00163     {
00164       return ERROR_CODE;
00165     }
00166   iov->iov_len = (pos == (amount - 1) ?  (size % CHUNK_SZ) : CHUNK_SZ);
00167   if (iov->iov_len == 0)
00168     {
00169       iov->iov_len = CHUNK_SZ;
00170     }
00171   chd_ptr[pos] = iov->iov_base;
00172   return SUCCESS_CODE;
00173 }
```

4.2.3.2 free_request()

```
void free_request (
              struct request * req,
              void ** children_ptr,
              uint64_t amount )
```

Definition at line 84 of file low_saurion.c.

```
00085 {
00086   if (children_ptr)
00087     {
00088       free (children_ptr);
00089       children_ptr = NULL;
00090     }
00091   for (uint64_t i = 0; i < amount; ++i)
00092     {
00093       free (req->iov[i].iov_base);
00094       req->iov[i].iov_base = NULL;
00095     }
00096   free (req);
00097   req = NULL;
00098   free (children_ptr);
00099   children_ptr = NULL;
00100 }
```

4.2.3.3 initialize_iovec()

```
int initialize_iovec (
              struct iovec * iov,
              const uint64_t amount,
              const uint64_t pos,
              const void * msg,
              const uint64_t size,
              const 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 105 of file low_saurion.c.

```
00107 {
00108   if (!iov || !iov->iov_base)
00109     {
00110       return ERROR_CODE;
00111     }
00112   if (msg)
00113     {
00114       uint64_t len = iov->iov_len;
00115       char *dest = (char *)iov->iov_base;
00116       char *orig = (char *)msg + pos * CHUNK_SZ;
00117       uint64_t cpy_sz = 0;
00118       if (h)
00119         {
00120           if (pos == 0)
00121             {
00122               uint64_t send_size = htonll (size);
00123               memcpy (dest, &send_size, sizeof (uint64_t));
00124               dest += sizeof (uint64_t);
00125               len -= sizeof (uint64_t);
00126             }
00127           else
00128             {
00129               orig -= sizeof (uint64_t);
00130             }
00131           if ((pos + 1) == amount)
00132             {
00133               --len;
00134               cpy_sz = (len < size ?  len :  size);
00135               dest[cpy_sz] = 0;
00136             }
00137         }
00138       cpy_sz = (len < size ?  len :  size);
00139       memcpy (dest, orig, cpy_sz);
00140       dest += cpy_sz;
00141       uint64_t rem = CHUNK_SZ - (dest - (char *)iov->iov_base);
00142       memset (dest, 0, rem);
00143     }
00144   else
00145     {
00146       memset ((char *)iov->iov_base, 0, CHUNK_SZ);
00147     }
00148   return SUCCESS_CODE;
00149 }
```

4.2.3.4 read_chunk()

```
int read_chunk (
            void ** dest,
            uint64_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 `uint64_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 `uint64_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 `uint64_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 `uint64_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.

Definition at line 678 of file low_saurion.c.

```
00679 {
00680   struct chunk_params p;
00681   p.req = req;
00682   p.dest = dest;
00683   p.len = len;
00684   if (p.req->iovec_count == 0)
00685     {
00686       return ERROR_CODE;
00687     }
00688
00689   p.max_iov_cont = calculate_max_iov_content (p.req);
00690   p.cont_sz = 0;
00691   p.cont_rem = 0;
00692   p.curr_iov = 0;
00693   p.curr_iov_off = 0;
00694   p.dest_off = 0;
00695   p.dest_ptr = NULL;
00696   if (!prepare_destination (&p))
00697     {
00698       return ERROR_CODE;
00699     }
00700
00701   uint8_t ok = 1UL;
00702   copy_data (&p, &ok);
00703
00704   if (validate_and_update (&p, ok))
00705     {
00706       return SUCCESS_CODE;
00707     }
00708   read_chunk_free (&p);
```

```
00709   return ERROR_CODE;
00710 }
```

4.2.3.5 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 833 of file low_saurion.c.

```
00834 {
00835   LOG_INIT (" ");
00836   struct saurion *p = (struct saurion *)malloc (sizeof (struct saurion));
00837   if (!p)
00838     {
00839       LOG_END (" ");
00840       return NULL;
00841     }
00842   int ret = 0;
00843   ret = pthread_mutex_init (&p->status_m, NULL);
00844   if (ret)
00845     {
00846       free (p);
00847       LOG_END (" ");
00848       return NULL;
00849     }
00850   ret = pthread_cond_init (&p->status_c, NULL);
00851   if (ret)
00852     {
00853       free (p);
00854       LOG_END (" ");
00855       return NULL;
00856     }
00857   p->m_rings
00858       = (pthread_mutex_t *)malloc (n_threads * sizeof (pthread_mutex_t));
00859   if (!p->m_rings)
00860     {
00861       free (p);
00862       LOG_END (" ");
00863       return NULL;
00864     }
00865   for (uint32_t i = 0; i < n_threads; ++i)
00866     {
00867       pthread_mutex_init (&(p->m_rings[i]), NULL);
00868     }
00869   p->ss = 0;
00870   n_threads = (n_threads < 2 ?  2 :  n_threads);
00871   n_threads = (n_threads > NUM_CORES ? NUM_CORES : n_threads);
00872   p->n_threads = n_threads;
00873   p->status = 0;
00874   p->list = NULL;
00875   p->cb.on_connected = NULL;
00876   p->cb.on_connected_arg = NULL;
00877   p->cb.on_readed = NULL;
00878   p->cb.on_readed_arg = NULL;
00879   p->cb.on_wrote = NULL;
```

```
00880   p->cb.on_wrote_arg = NULL;
00881   p->cb.on_closed = NULL;
00882   p->cb.on_closed_arg = NULL;
00883   p->cb.on_error = NULL;
00884   p->cb.on_error_arg = NULL;
00885   p->next = 0;
00886   p->efds = (int *)malloc (sizeof (int) * p->n_threads);
00887   if (!p->efds)
00888     {
00889       free (p->m_rings);
00890       free (p);
00891       LOG_END (" ");
00892       return NULL;
00893     }
00894   for (uint32_t i = 0; i < p->n_threads; ++i)
00895     {
00896       p->efds[i] = eventfd (0, EFD_NONBLOCK);
00897       if (p->efds[i] == ERROR_CODE)
00898         {
00899           for (uint32_t j = 0; j < i; ++j)
00900             {
00901               close (p->efds[j]);
00902             }
00903           free (p->efds);
00904           free (p->m_rings);
00905           free (p);
00906           LOG_END (" ");
00907           return NULL;
00908         }
00909     }
00910   p->rings
00911       = (struct io_uring *)malloc (sizeof (struct io_uring) * p->n_threads);
00912   if (!p->rings)
00913     {
00914       for (uint32_t j = 0; j < p->n_threads; ++j)
00915         {
00916           close (p->efds[j]);
00917         }
00918       free (p->efds);
00919       free (p->m_rings);
00920       free (p);
00921       LOG_END (" ");
00922       return NULL;
00923     }
00924   for (uint32_t i = 0; i < p->n_threads; ++i)
00925     {
00926       memset (&p->rings[i], 0, sizeof (struct io_uring));
00927       ret = io_uring_queue_init (SAURION_RING_SIZE, &p->rings[i], 0);
00928       if (ret)
00929         {
00930           for (uint32_t j = 0; j < p->n_threads; ++j)
00931             {
00932               close (p->efds[j]);
00933             }
00934           free (p->efds);
00935           free (p->rings);
00936           free (p->m_rings);
00937           free (p);
00938           LOG_END (" ");
00939           return NULL;
00940         }
00941     }
00942   p->pool = threadpool_create (p->n_threads);
00943   LOG_END (" ");
00944   return p;
00945 }
```

4.2.3.6 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 1187 of file low_saurion.c.

```
01188 {
01189   pthread_mutex_lock (&s->status_m);
01190   while (s->status > 0)
01191     {
01192       pthread_cond_wait (&s->status_c, &s->status_m);
01193     }
01194   pthread_mutex_unlock (&s->status_m);
01195   threadpool_destroy (s->pool);
01196   for (uint32_t i = 0; i < s->n_threads; ++i)
01197     {
01198       io_uring_queue_exit (&s->rings[i]);
01199       pthread_mutex_destroy (&s->m_rings[i]);
01200     }
01201   free (s->m_rings);
01202   list_free (&s->list);
01203   for (uint32_t i = 0; i < s->n_threads; ++i)
01204     {
01205       close (s->efds[i]);
01206     }
01207   free (s->efds);
01208   if (!s->ss)
01209     {
01210       close (s->ss);
01211     }
01212   free (s->rings);
01213   pthread_mutex_destroy (&s->status_m);
01214   pthread_cond_destroy (&s->status_c);
01215   free (s);
01216 }
```

4.2.3.7 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 1220 of file low_saurion.c.

```
01221 {
01222   add_write (s, fd, msg, next (s));
01223 }
```

4.2.3.8 saurion_set_socket()

```
int saurion_set_socket (
            const 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 787 of file low_saurion.c.

```
00788 {
00789   int sock = 0;
00790   struct sockaddr_in srv_addr;
00791
00792   sock = socket (PF_INET, SOCK_STREAM, 0);
00793   if (sock < 1)
00794     {
00795       return ERROR_CODE;
00796     }
00797
00798   int enable = 1;
00799   if (setsockopt (sock, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof (int)) < 0)
00800     {
00801       return ERROR_CODE;
00802     }
00803   struct timeval t_out;
00804   t_out.tv_sec = TIMEOUT_IDLE / 1000L;
00805   t_out.tv_usec = TIMEOUT_IDLE % 1000L;
00806   if (setsockopt (sock, SOL_SOCKET, SO_RCVTIMEO, &t_out, sizeof (t_out)) < 0)
00807     {
00808       return ERROR_CODE;
00809     }
00810
00811   memset (&srv_addr, 0, sizeof (srv_addr));
00812   srv_addr.sin_family = AF_INET;
00813   srv_addr.sin_port = htons (p);
00814   srv_addr.sin_addr.s_addr = htonl (INADDR_ANY);
00815
00816   if (bind (sock, (const struct sockaddr *)&srv_addr, sizeof (srv_addr)) < 0)
00817     {
00818       return ERROR_CODE;
00819     }
00820
00821   constexpr int num_queue = (ACCEPT_QUEUE > 0 ?  ACCEPT_QUEUE : SOMAXCONN);
00822   if (listen (sock, num_queue) < 0)
00823     {
00824       return ERROR_CODE;
00825     }
00826
00827   return sock;
00828 }
```

4.2.3.9 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 1145 of file low_saurion.c.

```
01146 {
01147   threadpool_init (s->pool);
01148   threadpool_add (s->pool, saurion_worker_master, s);
01149   struct saurion_wrapper *ss = NULL;
01150   for (uint32_t i = 1; i < s->n_threads; ++i)
01151     {
01152       ss = (struct saurion_wrapper *)malloc (sizeof (struct saurion_wrapper));
01153       if (!ss)
01154         {
01155           return ERROR_CODE;
01156         }
01157       ss->s = s;
01158       ss->sel = i;
01159       threadpool_add (s->pool, saurion_worker_slave, ss);
01160     }
01161   pthread_mutex_lock (&s->status_m);
01162   while (s->status < (int)s->n_threads)
01163     {
01164       pthread_cond_wait (&s->status_c, &s->status_m);
01165     }
01166   pthread_mutex_unlock (&s->status_m);
01167   return SUCCESS_CODE;
01168 }
```

4.2.3.10 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 1172 of file low_saurion.c.

```
01173 {
01174   uint64_t u = 1;
01175   for (uint32_t i = 0; i < s->n_threads; ++i)
01176     {
01177       while (write (s->efds[i], &u, sizeof (u)) < 0)
01178         {
01179           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
01180         }
01181     }
01182   threadpool_wait_empty (s->pool);
01183 }
```

4.2.3.11 set_request()

```
int set_request (
            struct request ** r,
            struct Node ** l,
            uint64_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 178 of file low_saurion.c.

```
00180 {
00181   uint64_t full_size = s;
00182   if (h)
00183     {
00184       full_size += (sizeof (uint64_t) + sizeof (uint8_t));
00185     }
00186   uint64_t amount = full_size / CHUNK_SZ;
00187   amount = amount + (full_size % CHUNK_SZ == 0 ?  0 :  1);
00188   struct request *temp = (struct request *)malloc (
00189       sizeof (struct request) + sizeof (struct iovec) * amount);
00190   if (!temp)
00191     {
00192       return ERROR_CODE;
00193     }
00194   if (!*r)
00195     {
00196       *r = temp;
```

```
00197       (*r)->prev = NULL;
00198       (*r)->prev_size = 0;
00199       (*r)->prev_remain = 0;
00200       (*r)->next_iov = 0;
00201       (*r)->next_offset = 0;
00202     }
00203   else
00204     {
00205       temp->client_socket = (*r)->client_socket;
00206       temp->event_type = (*r)->event_type;
00207       temp->prev = (*r)->prev;
00208       temp->prev_size = (*r)->prev_size;
00209       temp->prev_remain = (*r)->prev_remain;
00210       temp->next_iov = (*r)->next_iov;
00211       temp->next_offset = (*r)->next_offset;
00212       *r = temp;
00213     }
00214   struct request *req = *r;
00215   req->iovec_count = (int)amount;
00216   void **children_ptr = (void **)malloc (amount * sizeof (void *));
00217   if (!children_ptr)
00218     {
00219       free_request (req, children_ptr, 0);
00220       return ERROR_CODE;
00221     }
00222   for (uint64_t i = 0; i < amount; ++i)
00223     {
00224       if (!allocate_iovec (&req->iov[i], amount, i, full_size, children_ptr))
00225         {
00226           free_request (req, children_ptr, amount);
00227           return ERROR_CODE;
00228         }
00229       if (!initialize_iovec (&req->iov[i], amount, i, m, s, h))
00230         {
00231           free_request (req, children_ptr, amount);
00232           return ERROR_CODE;
00233         }
00234     }
00235   if (!list_insert (l, req, amount, children_ptr))
00236     {
00237       free_request (req, children_ptr, amount);
00238       return ERROR_CODE;
00239     }
00240   free (children_ptr);
00241   return SUCCESS_CODE;
00242 }
```

4.3 HighSaurion

Header file for the Saurion library.

Classes

- class Saurion

  A class for managing network connections with callback-based event handling.

4.3.1 Detailed Description

Header file for the Saurion library.

The Saurion library provides an abstraction for handling network connections with callback-based event handling.

4.3.1.0.1 Overview: The `Saurion` class manages a network socket and provides callbacks for various events:

- Connection established
- Data received
- Data sent
- Connection closed
- Errors

4.3.1.0.2 Class Diagram:

```
+--------------------+
|      Saurion       |
+--------------------+
| + init()           |
| + stop()           |
| + send(fd, msg)    |
| + on_connected()   |
| + on_readed()      |
| + on_wrote()       |
| + on_closed()      |
| + on_error()       |
+--------------------+
       | Uses
       v
+--------------------+
|  saurion struct    |
+--------------------+
```

4.3.1.0.3 Example Usage:

```
#include "saurion.hpp"
#include <iostream>
void on_connected(const int fd, void *arg) {
    std::cout « "Connected:  " « fd « std::endl;
}
void on_readed(const int fd, const void *const data, const int64_t len,
    void *arg) {
  std::cout « "Read data:  " « std::string((char *)data, len) « std::endl;
}
void on_closed(const int fd, void *arg) {
    std::cout « "Connection closed:  " « fd « std::endl;
}
int main() {
    Saurion server(4, 8080);
    server.on_connected(on_connected, nullptr)
          .on_readed(on_readed, nullptr)
          .on_closed(on_closed, nullptr);
    server.init();
    std::this_thread::sleep_for(std::chrono::seconds(10));
    server.stop();
    return 0;
}
```

Author

Israel

Date

2024

4.4 ThreadPool

A thread pool implementation for managing and executing tasks.

Classes

- struct threadpool

  Represents a thread pool.

Functions

- struct threadpool * threadpool_create (uint64_t num_threads)

  Creates a new thread pool with the specified number of threads.
- struct threadpool * threadpool_create_default (void)

  Creates a new thread pool with the default number of threads (equal to the number of CPU cores).
- void threadpool_init (struct threadpool *pool)

  Initializes the thread pool, starting the worker threads.
- void threadpool_add (struct threadpool *pool, void(*function)(void *), void *argument)

  Adds a task to the thread pool.
- void threadpool_stop (struct threadpool *pool)

  Stops all threads in the thread pool and prevents further tasks from being added.
- int threadpool_empty (struct threadpool *pool)

  Checks if the thread pool's task queue is empty.
- void threadpool_wait_empty (struct threadpool *pool)

  Waits until the task queue becomes empty.
- void threadpool_destroy (struct threadpool *pool)

  Destroys the thread pool, freeing all allocated resources.

4.4.1 Detailed Description

A thread pool implementation for managing and executing tasks.

This module provides functionality to manage a pool of threads that execute tasks in a synchronized manner.

4.4.1.0.1 Thread Pool Overview:

```
Threads in pool:  [T1] [T2] [T3] ...  [Tn]
Task Queue:
[Task(function=A, argument=X)] -> [Task(function=B, argument=Y)] -> NULL
```

4.4.1.0.2 Example Usage:

```
#include "threadpool.h"
#include <stdio.h>
void print_task(void *arg) {
    int *val = (int *)arg;
    printf("Task executed with value:  %d\n", *val);
}
int main() {
    struct threadpool *pool = threadpool_create_default();
    threadpool_init(pool);
    int data1 = 42, data2 = 24;
    threadpool_add(pool, print_task, &data1);
    threadpool_add(pool, print_task, &data2);
    threadpool_wait_empty(pool);
    threadpool_destroy(pool);
    return 0;
}
```

Author

Israel

Date

2024

4.4.2 Function Documentation

4.4.2.1 threadpool_add()

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

Adds a task to the thread pool.

Parameters

pool	Pointer to the thread pool.
function	Pointer to the function representing the task.
argument	Pointer to the argument to pass to the task function.

4.4.2.1.1 Diagram:

```
Before:
Task Queue:  Empty
After:
Task Queue:  [Task(function=A, argument=X)] -> NULL
```

Definition at line 167 of file threadpool.c.

```
00169 {
00170   LOG_INIT (" ");
00171   if (pool == NULL || function == NULL)
00172     {
00173       LOG_END (" ");
00174       return;
00175     }
00176
00177   struct task *new_task = malloc (sizeof (struct task));
00178   if (new_task == NULL)
00179     {
00180       LOG_END (" ");
00181       return;
00182     }
00183
00184   new_task->function = function;
00185   new_task->argument = argument;
00186   new_task->next = NULL;
00187
00188   pthread_mutex_lock (&pool->queue_lock);
00189
00190   if (pool->task_queue_head == NULL)
00191     {
00192       pool->task_queue_head = new_task;
00193       pool->task_queue_tail = new_task;
00194     }
00195   else
00196     {
00197       pool->task_queue_tail->next = new_task;
00198       pool->task_queue_tail = new_task;
00199     }
00200   pthread_cond_signal (&pool->queue_cond);
00201
00202   pthread_mutex_unlock (&pool->queue_lock);
00203   LOG_END (" ");
00204 }
```

4.4.2.2 threadpool_create()

```
struct threadpool * threadpool_create (
            uint64_t num_threads )
```

Creates a new thread pool with the specified number of threads.

Parameters

num_threads	The number of threads in the pool.

Returns

A pointer to the created thread pool, or `NULL` if creation fails.

4.4.2.2.1 Diagram:

```
Input:
num_threads = 4
Output:
ThreadPool:
  - Threads:  [T1, T2, T3, T4]
  - Task Queue:  Empty
```

Definition at line 30 of file threadpool.c.

```
00031 {
00032   LOG_INIT (" ");
00033   struct threadpool *pool = malloc (sizeof (struct threadpool));
00034   if (pool == NULL)
00035     {
00036       LOG_END (" ");
00037       return NULL;
00038     }
00039   if (num_threads < 3)
00040     {
00041       num_threads = 3;
00042     }
00043   if (num_threads > NUM_CORES)
00044     {
00045       num_threads = NUM_CORES;
00046     }
00047
00048   pool->num_threads = num_threads;
00049   pool->threads = malloc (sizeof (pthread_t) * num_threads);
00050   if (pool->threads == NULL)
00051     {
00052       free (pool);
00053       LOG_END (" ");
00054       return NULL;
00055     }
00056
00057   pool->task_queue_head = NULL;
00058   pool->task_queue_tail = NULL;
00059   pool->stop = FALSE;
00060   pool->started = FALSE;
00061
00062   if (pthread_mutex_init (&pool->queue_lock, NULL) != 0)
00063     {
00064       free (pool->threads);
00065       free (pool);
00066       LOG_END (" ");
00067       return NULL;
00068     }
00069
00070   if (pthread_cond_init (&pool->queue_cond, NULL) != 0)
00071     {
00072       pthread_mutex_destroy (&pool->queue_lock);
00073       free (pool->threads);
00074       free (pool);
00075       LOG_END (" ");
00076       return NULL;
00077     }
00078
00079   if (pthread_cond_init (&pool->empty_cond, NULL) != 0)
00080     {
00081       pthread_mutex_destroy (&pool->queue_lock);
00082       pthread_cond_destroy (&pool->queue_cond);
00083       free (pool->threads);
00084       free (pool);
00085       LOG_END (" ");
00086       return NULL;
00087     }
00088
00089   LOG_END (" ");
00090   return pool;
00091 }
```

4.4.2.3 threadpool_create_default()

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

Creates a new thread pool with the default number of threads (equal to the number of CPU cores).

Returns

A pointer to the created thread pool, or `NULL` if creation fails.

Definition at line 94 of file threadpool.c.

```
00095 {
00096   return threadpool_create (NUM_CORES);
00097 }
```

4.4.2.4 threadpool_destroy()

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

Destroys the thread pool, freeing all allocated resources.

Parameters

pool	Pointer to the thread pool to destroy.

4.4.2.4.1 Diagram: Before:

```
ThreadPool:
  - Threads:  [T1, T2, T3]
  - Task Queue:  [Task1] -> [Task2] -> NULL
After:
ThreadPool:  Destroyed
```

Definition at line 265 of file threadpool.c.

```
00266 {
00267   LOG_INIT (" ");
00268   if (pool == NULL)
00269     {
00270       LOG_END (" ");
00271       return;
00272     }
00273   threadpool_stop (pool);
00274
00275   pthread_mutex_destroy (&pool->queue_lock);
00276   pthread_cond_destroy (&pool->queue_cond);
00277   pthread_cond_destroy (&pool->empty_cond);
00278
00279   free (pool->threads);
00280   free (pool);
00281   LOG_END (" ");
00282 }
```

4.4.2.5 threadpool_empty()

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

Checks if the thread pool's task queue is empty.

Parameters

pool	Pointer to the thread pool.

Returns

`1` if the task queue is empty, `0` otherwise.

Definition at line 231 of file threadpool.c.

```
00232 {
00233   LOG_INIT (" ");
00234   if (pool == NULL)
00235     {
00236       LOG_END (" ");
00237       return TRUE;
00238     }
00239   pthread_mutex_lock (&pool->queue_lock);
00240   int empty = (pool->task_queue_head == NULL);
00241   pthread_mutex_unlock (&pool->queue_lock);
00242   LOG_END (" ");
00243   return empty;
00244 }
```

4.4.2.6 threadpool_init()

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

Initializes the thread pool, starting the worker threads.

Parameters

pool	Pointer to the thread pool to initialize.

Definition at line 144 of file threadpool.c.

```
00145 {
00146   LOG_INIT (" ");
00147   if (pool == NULL || pool->started)
00148     {
00149       LOG_END (" ");
00150       return;
00151     }
00152   for (uint64_t i = 0; i < pool->num_threads; i++)
00153     {
00154       if (pthread_create (&pool->threads[i], NULL, threadpool_worker,
00155                           (void *)pool)
00156           != 0)
00157         {
00158           pool->stop = TRUE;
00159           break;
00160         }
00161     }
00162   pool->started = TRUE;
00163   LOG_END (" ");
00164 }
```

4.4.2.7 threadpool_stop()

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

Stops all threads in the thread pool and prevents further tasks from being added.

Parameters

pool	Pointer to the thread pool to stop.

4.4.2.7.1 Diagram: Before:

```
Threads:  [Running T1, Running T2]
Task Queue:  [Task1] -> [Task2] -> NULL
After:
Threads:  Stopped
Task Queue:  Empty
```

Definition at line 207 of file threadpool.c.

```
00208 {
00209   LOG_INIT (" ");
00210   if (pool == NULL || !pool->started)
00211     {
00212       LOG_END (" ");
00213       return;
00214     }
00215   threadpool_wait_empty (pool);
00216
00217   pthread_mutex_lock (&pool->queue_lock);
00218   pool->stop = TRUE;
00219   pthread_cond_broadcast (&pool->queue_cond);
00220   pthread_mutex_unlock (&pool->queue_lock);
00221
00222   for (uint64_t i = 0; i < pool->num_threads; i++)
00223     {
00224       pthread_join (pool->threads[i], NULL);
00225     }
00226   pool->started = FALSE;
00227   LOG_END (" ");
00228 }
```

4.4.2.8 threadpool_wait_empty()

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

Waits until the task queue becomes empty.

Parameters

pool	Pointer to the thread pool.

Definition at line 247 of file threadpool.c.

```
00248 {
00249   LOG_INIT (" ");
00250   if (pool == NULL)
00251     {
00252       LOG_END (" ");
00253       return;
00254     }
00255   pthread_mutex_lock (&pool->queue_lock);
00256   while (pool->task_queue_head != NULL)
00257     {
00258       pthread_cond_wait (&pool->empty_cond, &pool->queue_lock);
00259     }
00260   pthread_mutex_unlock (&pool->queue_lock);
00261   LOG_END (" ");
00262 }
```

Chapter 5

Class Documentation

5.1 chunk_params Struct Reference

Collaboration diagram for chunk_params:

Public Attributes

- void ** dest
- void * dest_ptr
- uint64_t dest_off
- struct request * req
- uint64_t cont_sz
- uint64_t cont_rem
- uint64_t max_iov_cont
- uint64_t curr_iov
- uint64_t curr_iov_off
- uint64_t * len

5.1.1 Detailed Description

Definition at line 442 of file low_saurion.c.

5.1.2 Member Data Documentation

5.1.2.1 cont_rem

```
uint64_t chunk_params::cont_rem
```

Definition at line 449 of file low_saurion.c.

5.1.2.2 cont_sz

```
uint64_t chunk_params::cont_sz
```

Definition at line 448 of file low_saurion.c.

5.1.2.3 curr_iov

```
uint64_t chunk_params::curr_iov
```

Definition at line 451 of file low_saurion.c.

5.1.2.4 curr_iov_off

```
uint64_t chunk_params::curr_iov_off
```

Definition at line 452 of file low_saurion.c.

5.1.2.5 dest

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

Definition at line 444 of file low_saurion.c.

5.1.2.6 dest_off

```
uint64_t chunk_params::dest_off
```

Definition at line 446 of file low_saurion.c.

5.1.2.7 dest_ptr

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

Definition at line 445 of file low_saurion.c.

5.1.2.8 len

```
uint64_t* chunk_params::len
```

Definition at line 453 of file low_saurion.c.

5.1.2.9 max_iov_cont

```
uint64_t chunk_params::max_iov_cont
```

Definition at line 450 of file low_saurion.c.

5.1.2.10 req

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

Definition at line 447 of file low_saurion.c.

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

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

5.2 ClientInterface Class Reference

```
#include <client_interface.hpp>
```

Public Member Functions

- ClientInterface () noexcept
- ~ClientInterface ()
- ClientInterface (const ClientInterface &)=delete
- ClientInterface (ClientInterface &&)=delete
- ClientInterface & operator= (const ClientInterface &)=delete
- ClientInterface & operator= (ClientInterface &&)=delete
- void connect (const uint n)
- void disconnect ()
- void send (const uint n, const char *const msg, uint delay)
- uint64_t reads (const std::string &search) const
- void clean () const
- std::string getFifoPath () const
- int getPort () const

Private Attributes

- pid_t pid
- FILE * fifo
- std::string fifoname = set_fifoname ()
- int port = set_port ()

5.2.1 Detailed Description

Definition at line 13 of file client_interface.hpp.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 ClientInterface() [1/3]

```
ClientInterface::ClientInterface ( )  [explicit], [noexcept]
```

5.2.2.2 ~ClientInterface()

```
ClientInterface::~ClientInterface ( )
```

5.2.2.3 ClientInterface() [2/3]

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

5.2.2.4 ClientInterface() [3/3]

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

5.2.3 Member Function Documentation

5.2.3.1 clean()

```
void ClientInterface::clean ( ) const
```

5.2.3.2 connect()

```
void ClientInterface::connect (
            const uint n )
```

5.2.3.3 disconnect()

```
void ClientInterface::disconnect ( )
```

5.2.3.4 getFifoPath()

```
std::string ClientInterface::getFifoPath ( ) const
```

5.2.3.5 getPort()

```
int ClientInterface::getPort ( ) const
```

5.2.3.6 operator=() [1/2]

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

5.2.3.7 operator=() [2/2]

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

5.2.3.8 reads()

```
uint64_t ClientInterface::reads (
            const std::string & search ) const
```

5.2.3.9 send()

```
void ClientInterface::send (
            const uint n,
            const char *const msg,
            uint delay )
```

5.2.4 Member Data Documentation

5.2.4.1 fifo

```
FILE* ClientInterface::fifo  [private]
```

Definition at line 36 of file client_interface.hpp.

5.2.4.2 fifoname

```
std::string ClientInterface::fifoname = set_fifoname ()  [private]
```

Definition at line 37 of file client_interface.hpp.

5.2.4.3 pid

```
pid_t ClientInterface::pid  [private]
```

Definition at line 35 of file client_interface.hpp.

5.2.4.4 port

```
int ClientInterface::port = set_port ()  [private]
```

Definition at line 38 of file client_interface.hpp.

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

- /__w/saurion/saurion/include/client_interface.hpp

5.3 Node Struct Reference

Represents a node in the linked list.

`#include <linked_list.h>`

Collaboration diagram for Node:

Public Attributes

- void * ptr
- uint64_t size
- struct Node ** children
- struct Node * next

5.3.1 Detailed Description

Represents a node in the linked list.

Each node stores a pointer, a size, an array of child nodes, and a pointer to the next node in the list.

Definition at line 7 of file linked_list.c.

5.3.2 Member Data Documentation

5.3.2.1 children

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

Definition at line 11 of file linked_list.c.

5.3.2.2 next

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

Definition at line 12 of file linked_list.c.

5.3.2.3 ptr

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

Definition at line 9 of file linked_list.c.

5.3.2.4 size

```
uint64_t Node::size
```

Definition at line 10 of file linked_list.c.

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

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

5.4 request Struct Reference

Public Attributes

- void * prev
- uint64_t prev_size
- uint64_t prev_remain
- uint64_t next_iov
- uint64_t next_offset
- int event_type
- uint64_t iovec_count
- int client_socket
- struct iovec iov [ ]

5.4.1 Detailed Description

Definition at line 22 of file low_saurion.c.

5.4.2 Member Data Documentation

5.4.2.1 client_socket

```
int request::client_socket
```

Definition at line 31 of file low_saurion.c.

5.4.2.2 event_type

```
int request::event_type
```

Definition at line 29 of file low_saurion.c.

5.4.2.3 iov

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

Definition at line 32 of file low_saurion.c.

5.4.2.4 iovec_count

```
uint64_t request::iovec_count
```

Definition at line 30 of file low_saurion.c.

5.4.2.5 next_iov

```
uint64_t request::next_iov
```

Definition at line 27 of file low_saurion.c.

5.4.2.6 next_offset

```
uint64_t request::next_offset
```

Definition at line 28 of file low_saurion.c.

5.4.2.7 prev

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

Definition at line 24 of file low_saurion.c.

5.4.2.8 prev_remain

```
uint64_t request::prev_remain
```

Definition at line 26 of file low_saurion.c.

5.4.2.9 prev_size

```
uint64_t request::prev_size
```

Definition at line 25 of file low_saurion.c.

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

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

5.5 saurion Struct Reference

Main structure for managing io_uring and socket events.

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

Collaboration diagram for saurion:

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
- struct saurion_callbacks cb

5.5.1 Detailed Description

Main structure for managing io_uring and socket events.

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

Definition at line 215 of file low_saurion.h.

5.5.2 Member Data Documentation

5.5.2.1 cb

```
struct saurion_callbacks saurion::cb
```

Definition at line 240 of file low_saurion.h.

5.5.2.2 efds

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

Eventfd descriptors used for internal signaling between threads.

Definition at line 224 of file low_saurion.h.

5.5.2.3 list

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

Linked list for storing active requests.

Definition at line 226 of file low_saurion.h.

5.5.2.4 m_rings

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

Array of mutexes to protect the io_uring rings.

Definition at line 220 of file low_saurion.h.

5.5.2.5 n_threads

```
uint32_t saurion::n_threads
```

Number of threads in the thread pool.

Definition at line 236 of file low_saurion.h.

5.5.2.6 next

```
uint32_t saurion::next
```

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

Definition at line 238 of file low_saurion.h.

5.5.2.7 pool

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

Thread pool for executing tasks in parallel.

Definition at line 234 of file low_saurion.h.

5.5.2.8 rings

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

Array of io_uring structures for managing the event queue.

Definition at line 218 of file low_saurion.h.

5.5.2.9 ss

```
int saurion::ss
```

Server socket descriptor for accepting connections.

Definition at line 222 of file low_saurion.h.

5.5.2.10 status

```
int saurion::status
```

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

Definition at line 232 of file low_saurion.h.

5.5.2.11 status_c

```
pthread_cond_t saurion::status_c
```

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

Definition at line 230 of file low_saurion.h.

5.5.2.12 status_m

```
pthread_mutex_t saurion::status_m
```

Mutex to protect the state of the structure.

Definition at line 228 of file low_saurion.h.

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

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

5.6 Saurion Class Reference

A class for managing network connections with callback-based event handling.

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

Collaboration diagram for Saurion:

Public Types

- using ConnectedCb = void(∗)(const int, void ∗)

  Callback type for connection events.
- using ReadedCb = void(∗)(const int, const void ∗const, const int64_t, void ∗)

  Callback type for data received events.
- using WroteCb = void(∗)(const int, void ∗)

  Callback type for data sent events.
- using ClosedCb = void(∗)(const int, void ∗)

  Callback type for connection closed events.
- using ErrorCb = void(∗)(const int, const char ∗const, const int64_t, void ∗)

  Callback type for error events.

Public Member Functions

- Saurion (const uint32_t thds, const int sck) noexcept

  Constructs a `Saurion` instance.
- ~Saurion ()

  Destroys the `Saurion` instance, releasing resources.
- Saurion (const Saurion &)=delete
- Saurion (Saurion &&)=delete
- Saurion & operator= (const Saurion &)=delete
- Saurion & operator= (Saurion &&)=delete
- void init ()

  Initializes the server and starts listening for connections.
- void stop () const noexcept

  Stops the server and all associated threads.
- Saurion * on_connected (ConnectedCb ncb, void *arg) noexcept

  Sets the callback for connection events.
- Saurion * on_readed (ReadedCb ncb, void *arg) noexcept

  Sets the callback for data received events.
- Saurion * on_wrote (WroteCb ncb, void *arg) noexcept

  Sets the callback for data sent events.
- Saurion * on_closed (ClosedCb ncb, void *arg) noexcept

  Sets the callback for connection closed events.
- Saurion * on_error (ErrorCb ncb, void *arg) noexcept

  Sets the callback for error events.
- void send (const int fd, const char *const msg) noexcept

  Sends a message to the specified file descriptor.

Private Attributes

- struct saurion * s

  Pointer to the underlying `saurion` structure.

5.6.1 Detailed Description

A class for managing network connections with callback-based event handling.

The `Saurion` class encapsulates a network socket and provides methods for initializing, stopping, and sending data, as well as setting up event callbacks.

Definition at line 93 of file saurion.hpp.

5.6.2 Member Typedef Documentation

5.6.2.1 ClosedCb

`Saurion::ClosedCb`

Callback type for connection closed events.

Parameters

fd	File descriptor of the closed socket.
arg	User-defined argument.

Definition at line 126 of file saurion.hpp.

5.6.2.2 ConnectedCb

`Saurion::ConnectedCb`

Callback type for connection events.

Parameters

fd	File descriptor for the connected socket.
arg	User-defined argument.

Definition at line 102 of file saurion.hpp.

5.6.2.3 ErrorCb

`Saurion::ErrorCb`

Callback type for error events.

Parameters

fd	File descriptor of the socket where the error occurred.
msg	Error message.
len	Length of the error message.
arg	User-defined argument.

Definition at line 135 of file saurion.hpp.

5.6.2.4 ReadedCb

`Saurion::ReadedCb`

Callback type for data received events.

Parameters

fd	File descriptor of the socket.
data	Pointer to the received data.
len	Length of the received data.
arg	User-defined argument.

Definition at line 111 of file saurion.hpp.

5.6.2.5 WroteCb

`Saurion::WroteCb`

Callback type for data sent events.

Parameters

fd	File descriptor of the socket.
arg	User-defined argument.

Definition at line 119 of file saurion.hpp.

5.6.3 Constructor & Destructor Documentation

5.6.3.1 Saurion() [1/3]

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

Constructs a `Saurion` instance.

Parameters

thds	Number of threads for handling connections.
sck	Listening socket file descriptor.

Definition at line 7 of file saurion.cpp.

```
00008 {
00009   this->s = saurion_create (thds);
00010   if (!this->s)
00011     {
00012       return;
00013     }
00014   this->s->ss = sck;
00015 }
```

5.6.3.2 ~Saurion()

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

Destroys the Saurion instance, releasing resources.

Definition at line 17 of file saurion.cpp.

```
00018 {
00019   close (s->ss);
00020   saurion_destroy (this->s);
00021 }
```

5.6.3.3 Saurion() [2/3]

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

5.6.3.4 Saurion() [3/3]

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

5.6.4 Member Function Documentation

5.6.4.1 init()

```
void Saurion::init ( )
```

Initializes the server and starts listening for connections.

Definition at line 24 of file saurion.cpp.

```
00025 {
00026   if (!saurion_start (this->s))
00027     {
00028       throw std::runtime_error ("Error on saurion start");
00029     }
00030 }
```

5.6.4.2 on_closed()

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

Sets the callback for connection closed events.

Parameters

ncb	The callback function.
arg	User-defined argument for the callback.

Returns

Pointer to the `Saurion` instance for chaining.

Definition at line 63 of file saurion.cpp.

```
00064 {
00065   s->cb.on_closed = ncb;
00066   s->cb.on_closed_arg = arg;
00067   return this;
00068 }
```

5.6.4.3 on_connected()

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

Sets the callback for connection events.

Parameters

ncb	The callback function.
arg	User-defined argument for the callback.

Returns

Pointer to the `Saurion` instance for chaining.

Definition at line 39 of file saurion.cpp.

```
00040 {
00041   s->cb.on_connected = ncb;
00042   s->cb.on_connected_arg = arg;
00043   return this;
00044 }
```

5.6.4.4 on_error()

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

Sets the callback for error events.

Parameters

ncb	The callback function.
arg	User-defined argument for the callback.

Returns

Pointer to the `Saurion` instance for chaining.

Definition at line 71 of file saurion.cpp.

```
00072 {
00073   s->cb.on_error = ncb;
00074   s->cb.on_error_arg = arg;
00075   return this;
00076 }
```

5.6.4.5 on_readed()

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

Sets the callback for data received events.

Parameters

ncb	The callback function.
arg	User-defined argument for the callback.

Returns

Pointer to the `Saurion` instance for chaining.

Definition at line 47 of file saurion.cpp.

```
00048 {
00049   s->cb.on_readed = ncb;
00050   s->cb.on_readed_arg = arg;
00051   return this;
00052 }
```

5.6.4.6 on_wrote()

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

Sets the callback for data sent events.

Parameters

ncb	The callback function.
arg	User-defined argument for the callback.

Returns

Pointer to the `Saurion` instance for chaining.

Definition at line 55 of file saurion.cpp.

```
00056 {
00057   s->cb.on_wrote = ncb;
00058   s->cb.on_wrote_arg = arg;
00059   return this;
00060 }
```

5.6.4.7 operator=() [1/2]

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

5.6.4.8 operator=() [2/2]

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

5.6.4.9 send()

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

Sends a message to the specified file descriptor.

Parameters

fd	File descriptor to send the message to.
msg	Pointer to the message to send.

Definition at line 79 of file saurion.cpp.

```
00080 {
00081   saurion_send (this->s, fd, msg);
00082 }
```

5.6.4.10 stop()

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

Stops the server and all associated threads.

Definition at line 33 of file saurion.cpp.

```
00034 {
00035   saurion_stop (this->s);
00036 }
```

5.6.5 Member Data Documentation

5.6.5.1 s

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

Pointer to the underlying `saurion` structure.

Definition at line 207 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

5.7 saurion_callbacks Struct Reference

Structure containing callback functions to handle socket events.

`#include <low_saurion.h>`

Public Attributes

- void(∗ on_connected)(const int fd, void ∗arg)
  Callback for handling new connections.
- void ∗ on_connected_arg
- void(∗ on_readed)(const int fd, const void ∗const content, const int64_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 int64_t len, void ∗arg)
  Callback for handling error events.
- void ∗ on_error_arg

5.7.1 Detailed Description

Structure containing callback functions to handle socket events.

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

Definition at line 150 of file low_saurion.h.

5.7.2 Member Data Documentation

5.7.2.1 on_closed

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

Callback for handling socket closures.

Parameters

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

Definition at line 190 of file low_saurion.h.

5.7.2.2 on_closed_arg

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

Additional argument for the close callback.

Definition at line 192 of file low_saurion.h.

5.7.2.3 on_connected

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

Callback for handling new connections.

Parameters

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

Definition at line 158 of file low_saurion.h.

5.7.2.4 on_connected_arg

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

Additional argument for the connection callback.

Definition at line 160 of file low_saurion.h.

5.7.2.5 on_error

```
void(* saurion_callbacks::on_error) (const int fd, const char *const content, const int64_↩
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 202 of file low_saurion.h.

5.7.2.6 on_error_arg

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

Additional argument for the error callback.

Definition at line 205 of file low_saurion.h.

5.7.2.7 on_readed

```
void(* saurion_callbacks::on_readed) (const int fd, const void *const content, const int64_↩
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 170 of file low_saurion.h.

5.7.2.8 on_readed_arg

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

Additional argument for the read callback.

Definition at line 173 of file low_saurion.h.

5.7.2.9 on_wrote

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

Callback for handling write events.

Parameters

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

Definition at line 181 of file low_saurion.h.

5.7.2.10 on_wrote_arg

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

Additional argument for the write callback.

Definition at line 182 of file low_saurion.h.

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

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

5.8 saurion_wrapper Struct Reference

Collaboration diagram for saurion_wrapper:

Public Attributes

- struct saurion * s
- uint32_t sel

5.8.1 Detailed Description

Definition at line 40 of file low_saurion.c.

5.8.2 Member Data Documentation

5.8.2.1 s

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

Definition at line 42 of file low_saurion.c.

5.8.2.2 sel

```
uint32_t saurion_wrapper::sel
```

Definition at line 43 of file low_saurion.c.

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

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

5.9 task Struct Reference

Collaboration diagram for task:

Public Attributes

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

5.9.1 Detailed Description

Definition at line 9 of file threadpool.c.

5.9.2 Member Data Documentation

5.9.2.1 argument

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

Definition at line 12 of file threadpool.c.

5.9.2.2 function

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

Definition at line 11 of file threadpool.c.

5.9.2.3 next

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

Definition at line 13 of file threadpool.c.

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

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

5.10 threadpool Struct Reference

Represents a thread pool.

```
#include <threadpool.h>
```

Collaboration diagram for threadpool:

Public Attributes

- pthread_t * threads
- uint64_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

5.10.1 Detailed Description

Represents a thread pool.

The thread pool manages a fixed number of worker threads and a queue of tasks.

Definition at line 16 of file threadpool.c.

5.10.2 Member Data Documentation

5.10.2.1 empty_cond

```
pthread_cond_t threadpool::empty_cond
```

Definition at line 24 of file threadpool.c.

5.10.2.2 num_threads

```
uint64_t threadpool::num_threads
```

Definition at line 19 of file threadpool.c.

5.10.2.3 queue_cond

```
pthread_cond_t threadpool::queue_cond
```

Definition at line 23 of file threadpool.c.

5.10.2.4 queue_lock

```
pthread_mutex_t threadpool::queue_lock
```

Definition at line 22 of file threadpool.c.

5.10.2.5 started

```
int threadpool::started
```

Definition at line 26 of file threadpool.c.

5.10.2.6 stop

```
int threadpool::stop
```

Definition at line 25 of file threadpool.c.

5.10.2.7 task_queue_head

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

Definition at line 20 of file threadpool.c.

5.10.2.8 task_queue_tail

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

Definition at line 21 of file threadpool.c.

5.10.2.9 threads

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

Definition at line 18 of file threadpool.c.

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

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

Chapter 6

File Documentation

6.1 /__w/saurion/saurion/include/client_interface.hpp File Reference

```
#include <cstdint>
#include <string>
```

Include dependency graph for client_interface.hpp:

Classes

- class ClientInterface

Functions

- int set_port ()
- std::string set_fifoname ()

6.1.1 Function Documentation

6.1.1.1 set_fifoname()

```
std::string set_fifoname ( )
```

6.1.1.2 set_port()

```
int set_port ( )
```

6.2 client_interface.hpp

Go to the documentation of this file.

```
00001 #ifndef CLIENT_INTERFACE_HPP
00002 #define CLIENT_INTERFACE_HPP
00003
00004 #include <cstdint> // for uint64_t
00005 #include <string>  // for string
00006
00007 // set_port
00008 int set_port ();
00009
00010 // set_fifoname
00011 std::string set_fifoname ();
00012
00013 class ClientInterface
00014 {
00015 public:
00016   explicit ClientInterface () noexcept;
00017   ~ClientInterface ();
00018
00019   ClientInterface (const ClientInterface &) = delete;
00020   ClientInterface (ClientInterface &&) = delete;
00021   ClientInterface &operator= (const ClientInterface &) = delete;
00022   ClientInterface &operator= (ClientInterface &&) = delete;
00023
00024   void connect (const uint n);
00025   void disconnect ();
00026
00027   void send (const uint n, const char *const msg, uint delay);
00028   uint64_t reads (const std::string &search) const;
00029   void clean () const;
00030
00031   std::string getFifoPath () const;
00032   int getPort () const;
00033
00034 private:
00035   pid_t pid;
00036   FILE *fifo;
00037   std::string fifoname = set_fifoname ();
00038   int port = set_port ();
00039 };
00040
00041 #endif // !CLIENT_INTERFACE_HPP
```

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

```
#include <stdint.h>
#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, const uint64_t amount, void **children)

  Inserts a new node into the linked list.
- void list_delete_node (struct Node **head, const void *const ptr)

  Deletes a node from the linked list.
- void list_free (struct Node **head)

  Frees the entire linked list.

6.4 linked_list.h

Go to the documentation of this file.

```
00001
00048 #ifndef LINKED_LIST_H
00049 #define LINKED_LIST_H
00050
00051 #include <stdint.h> // for uint64_t
00052
00053 #ifdef __cplusplus
00054 extern "C"
00055 {
00056 #endif
00057
00058 #include <stddef.h>
00059
00067   struct Node;
00068
00093   [[nodiscard]]
00094   int list_insert (struct Node **head, void *ptr, const uint64_t amount,
00095                    void **children);
00096
00118   void list_delete_node (struct Node **head, const void *const ptr);
00119
00140   void list_free (struct Node **head);
00141
00142 #ifdef __cplusplus
00143 }
00144 #endif
00145
00146 #endif // !LINKED_LIST_H
00147
```

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

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

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

Classes

- struct saurion_callbacks

  Structure containing callback functions to handle socket events.
- 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 (const 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 int64_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 int64_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
- struct saurion_callbacks cb

6.5.1 Variable Documentation

6.5.1.1 cb

```
struct saurion_callbacks cb
```

Definition at line 23 of file low_saurion.h.

6.5.1.2 efds

```
int* efds
```

Eventfd descriptors used for internal signaling between threads.

Definition at line 7 of file low_saurion.h.

6.5.1.3 list

```
struct Node* list
```

Linked list for storing active requests.

Definition at line 9 of file low_saurion.h.

6.5.1.4 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.

6.5.1.5 n_threads

```
uint32_t n_threads
```

Number of threads in the thread pool.

Definition at line 19 of file low_saurion.h.

6.5.1.6 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.

6.5.1.7 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.

6.5.1.8 on_closed_arg

```
void* on_closed_arg
```

Additional argument for the close callback.

Definition at line 40 of file low_saurion.h.

6.5.1.9 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.

6.5.1.10 on_connected_arg

```
void* on_connected_arg
```

Additional argument for the connection callback.

Definition at line 8 of file low_saurion.h.

6.5.1.11 on_error

```
void(* on_error) (const int fd, const char *const content, const int64_t len, void *arg) (
            const int fd,
            const char *const content,
            const int64_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.

6.5.1.12 on_error_arg

```
void* on_error_arg
```

Additional argument for the error callback.

Definition at line 53 of file low_saurion.h.

6.5.1.13 on_readed

```
void(* on_readed) (const int fd, const void *const content, const int64_t len, void *arg) (
            const int fd,
            const void *const content,
            const int64_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.

6.5.1.14 on_readed_arg

```
void* on_readed_arg
```

Additional argument for the read callback.

Definition at line 21 of file low_saurion.h.

6.5.1.15 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.

6.5.1.16 on_wrote_arg

```
void* on_wrote_arg
```

Additional argument for the write callback.

Definition at line 30 of file low_saurion.h.

6.5.1.17 pool

```
struct threadpool* pool
```

Thread pool for executing tasks in parallel.

Definition at line 17 of file low_saurion.h.

6.5.1.18 rings

```
struct io_uring* rings
```

Array of io_uring structures for managing the event queue.

Definition at line 1 of file low_saurion.h.

6.5.1.19 ss

```
int ss
```

Server socket descriptor for accepting connections.

Definition at line 5 of file low_saurion.h.

6.5.1.20 status

```
int status
```

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

Definition at line 15 of file low_saurion.h.

6.5.1.21 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.

6.5.1.22 status_m

```
pthread_mutex_t status_m
```

Mutex to protect the state of the structure.

Definition at line 11 of file low_saurion.h.

6.6 low_saurion.h

Go to the documentation of this file.

```
00001
00105 #ifndef LOW_SAURION_H
00106 #define LOW_SAURION_H
00107
00108 #define _POSIX_C_SOURCE 200809L
00109
00110 #include <pthread.h> // for pthread_mutex_t, pthread_cond_t
00111 #include <stdint.h>  // for uint32_t, int64_t
00112
00113 // TODO: añadir métodos de backpressure, por ver conversacion de ChatGPT
00114
00115 #ifdef __cplusplus
00116 extern "C"
00117 {
00118 #endif
00119
00141 #define PACKING_SZ 32
00150   struct saurion_callbacks
00151   {
00158     void (*on_connected) (const int fd, void *arg);
00160     void *on_connected_arg;
00161
00170     void (*on_readed) (const int fd, const void *const content,
00171                        const int64_t len, void *arg);
00173     void *on_readed_arg;
00174
00181     void (*on_wrote) (const int fd, void *arg);
00182     void *on_wrote_arg;
00190     void (*on_closed) (const int fd, void *arg);
00192     void *on_closed_arg;
00193
00202     void (*on_error) (const int fd, const char *const content,
00203                       const int64_t len, void *arg);
00205     void *on_error_arg;
00206   } __attribute__ ((aligned (PACKING_SZ)));
00207
00215   struct saurion
00216   {
00218     struct io_uring *rings;
00220     pthread_mutex_t *m_rings;
00222     int ss;
00224     int *efds;
00226     struct Node *list;
00228     pthread_mutex_t status_m;
00230     pthread_cond_t status_c;
00232     int status;
00234     struct threadpool *pool;
00236     uint32_t n_threads;
00238     uint32_t next;
00239
00240     struct saurion_callbacks cb;
00241   } __attribute__ ((aligned (PACKING_SZ)));
00242
00252   int saurion_set_socket (const int p);
00253
00266   [[nodiscard]]
00267   struct saurion *saurion_create (uint32_t n_threads);
00268
00281   [[nodiscard]]
00282   int saurion_start (struct saurion *s);
00283
00294   void saurion_stop (const struct saurion *s);
00295
00308   void saurion_destroy (struct saurion *s);
00309
00322   void saurion_send (struct saurion *s, const int fd, const char *const msg);
00323
00324 #ifdef __cplusplus
00325 }
00326 #endif
00327
00328 #endif // !LOW_SAURION_H
00329
```

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

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

Include dependency graph for low_saurion_secret.h:

Functions

- int allocate_iovec (struct iovec *iov, const uint64_t amount, const uint64_t pos, const uint64_t size, void **chd_ptr)
- int initialize_iovec (struct iovec *iov, const uint64_t amount, const uint64_t pos, const void *msg, const uint64_t size, const uint8_t h)

  Initializes a specified `iovec` structure with a message fragment.
- int set_request (struct request **r, struct Node **l, uint64_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, uint64_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, uint64_t amount)

6.8 low_saurion_secret.h

Go to the documentation of this file.

```
00001 #ifndef LOW_SAURION_SECRET_H
00002 #define LOW_SAURION_SECRET_H
00003
00004 #include <bits/types/struct_iovec.h> // for struct iovec
00005 #include <stdint.h>                  // for uint64_t, uint8_t
00006
00007 #ifdef __cplusplus
00008 extern "C"
00009 {
00010 #endif
00015 #pragma GCC diagnostic push
00016 #pragma GCC diagnostic ignored "-Wpedantic"
00017   struct request
00018   {
00019     void *prev;
00020     uint64_t prev_size;
00021     uint64_t prev_remain;
00022     uint64_t next_iov;
00023     uint64_t next_offset;
00024     int event_type;
00025     uint64_t iovec_count;
00026     int client_socket;
00027     struct iovec iov[];
00028   };
00029 #pragma GCC diagnostic pop
00030
00031   struct Node;
00066   [[nodiscard]]
00067   int allocate_iovec (struct iovec *iov, const uint64_t amount,
00068                       const uint64_t pos, const uint64_t size, void **chd_ptr);
00069
00105   [[nodiscard]]
00106   int initialize_iovec (struct iovec *iov, const uint64_t amount,
00107                         const uint64_t pos, const void *msg,
00108                         const uint64_t size, const uint8_t h);
00109
00144   [[nodiscard]]
00145   int set_request (struct request **r, struct Node **l, uint64_t s,
00146                    const void *m, uint8_t h);
00147
00185   [[nodiscard]]
00186   int read_chunk (void **dest, uint64_t *len, struct request *const req);
00187
00188   void free_request (struct request *req, void **children_ptr,
00189                      uint64_t amount);
00193 #ifdef __cplusplus
00194 }
00195 #endif
00196
00197 #endif // !LOW_SAURION_SECRET_H
```

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

```
#include <cstdint>
#include <stdint.h>
```

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

Classes

- class Saurion

  A class for managing network connections with callback-based event handling.

6.10 saurion.hpp

Go to the documentation of this file.

```
00001
00079 #ifndef SAURION_HPP
00080 #define SAURION_HPP
00081
00082 #include <cstdint>
00083 #include <stdint.h> // for uint32_t, int64_t
00084
00093 class Saurion
00094 {
00095 public:
00102   using ConnectedCb = void (*) (const int, void *);
00111   using ReadedCb
00112       = void (*) (const int, const void *const, const int64_t, void *);
00119   using WroteCb = void (*) (const int, void *);
00126   using ClosedCb = void (*) (const int, void *);
00135   using ErrorCb
00136       = void (*) (const int, const char *const, const int64_t, void *);
00137
00143   explicit Saurion (const uint32_t thds, const int sck) noexcept;
00147   ~Saurion ();
00148
00149   Saurion (const Saurion &) = delete;
00150   Saurion (Saurion &&) = delete;
00151   Saurion &operator= (const Saurion &) = delete;
00152   Saurion &operator= (Saurion &&) = delete;
00153
00157   void init ();
00161   void stop () const noexcept;
00162
00169   Saurion *on_connected (ConnectedCb ncb, void *arg) noexcept;
00176   Saurion *on_readed (ReadedCb ncb, void *arg) noexcept;
00183   Saurion *on_wrote (WroteCb ncb, void *arg) noexcept;
00190   Saurion *on_closed (ClosedCb ncb, void *arg) noexcept;
00197   Saurion *on_error (ErrorCb ncb, void *arg) noexcept;
00198
00204   void send (const int fd, const char *const msg) noexcept;
00205
00206 private:
00207   struct saurion *s;
00208 };
00209
00210 #endif // !SAURION_HPP
00211
```

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

```
#include <stdint.h>
```

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

Functions

- struct threadpool * threadpool_create (uint64_t num_threads)

  Creates a new thread pool with the specified number of threads.
- struct threadpool * threadpool_create_default (void)

  Creates a new thread pool with the default number of threads (equal to the number of CPU cores).
- void threadpool_init (struct threadpool *pool)

  Initializes the thread pool, starting the worker threads.
- void threadpool_add (struct threadpool *pool, void(*function)(void *), void *argument)

  Adds a task to the thread pool.
- void threadpool_stop (struct threadpool *pool)

  Stops all threads in the thread pool and prevents further tasks from being added.
- int threadpool_empty (struct threadpool *pool)

  Checks if the thread pool's task queue is empty.
- void threadpool_wait_empty (struct threadpool *pool)

  Waits until the task queue becomes empty.
- void threadpool_destroy (struct threadpool *pool)

  Destroys the thread pool, freeing all allocated resources.

6.12 threadpool.h

Go to the documentation of this file.

```
00001
00048 #ifndef THREADPOOL_H
00049 #define THREADPOOL_H
00050
00051 #include <stdint.h> // for uint64_t
00052
00053 #ifdef __cplusplus
00054 extern "C"
00055 {
00056 #endif
00057
00065   struct threadpool;
00066
00084   struct threadpool *threadpool_create (uint64_t num_threads);
00085
00092   struct threadpool *threadpool_create_default (void);
00093
00099   void threadpool_init (struct threadpool *pool);
00100
00117   void threadpool_add (struct threadpool *pool, void (*function) (void *),
00118                        void *argument);
00119
00137   void threadpool_stop (struct threadpool *pool);
00138
00145   int threadpool_empty (struct threadpool *pool);
00146
00152   void threadpool_wait_empty (struct threadpool *pool);
00153
00170   void threadpool_destroy (struct threadpool *pool);
00171
00172 #ifdef __cplusplus
00173 }
00174 #endif
00175
00176 #endif // !THREADPOOL_H
00177
```

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

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

Include dependency graph for linked_list.c:

Classes

- struct Node

  Represents a node in the linked list.

Functions

- struct Node * create_node (void *ptr, const uint64_t amount, void *const *children)
- int list_insert (struct Node **head, void *ptr, const uint64_t amount, void **children)

  Inserts a new node into the linked list.
- void free_node (struct Node *current)
- void list_delete_node (struct Node **head, const void *const ptr)

  Deletes a node from the linked list.
- void list_free (struct Node **head)

  Frees the entire linked list.

Variables

- pthread_mutex_t list_mutex = PTHREAD_MUTEX_INITIALIZER

6.13.1 Function Documentation

6.13.1.1 create_node()

```
struct Node * create_node (
            void * ptr,
            const uint64_t amount,
            void *const * children )
```

Definition at line 20 of file linked_list.c.

```
00021 {
00022   struct Node *new_node = (struct Node *)malloc (sizeof (struct Node));
00023   if (!new_node)
00024     {
00025       return NULL;
00026     }
00027   new_node->ptr = ptr;
00028   new_node->size = amount;
00029   new_node->children = NULL;
00030   if (amount <= 0)
00031     {
00032       new_node->next = NULL;
00033       return new_node;
00034     }
00035   new_node->children
00036       = (struct Node **)malloc (sizeof (struct Node *) * amount);
00037   if (!new_node->children)
00038     {
00039       free (new_node);
00040       return NULL;
00041     }
00042   for (uint64_t i = 0; i < amount; ++i)
00043     {
00044       new_node->children[i] = (struct Node *)malloc (sizeof (struct Node));
00045
00046       if (!new_node->children[i])
00047         {
00048           for (uint64_t j = 0; j < i; ++j)
```

```
00049             {
00050               free (new_node->children[j]);
00051             }
00052           free (new_node);
00053           return NULL;
00054         }
00055     }
00056   for (uint64_t i = 0; i < amount; ++i)
00057     {
00058       new_node->children[i]->size = 0;
00059       new_node->children[i]->next = NULL;
00060       new_node->children[i]->ptr = children[i];
00061       new_node->children[i]->children = NULL;
00062     }
00063   new_node->next = NULL;
00064   return new_node;
00065 }
```

6.13.1.2 free_node()

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

Definition at line 97 of file linked_list.c.

```
00098 {
00099   if (current->size > 0)
00100     {
00101       for (uint64_t i = 0; i < current->size; ++i)
00102         {
00103           free (current->children[i]->ptr);
00104           free (current->children[i]);
00105         }
00106       free (current->children);
00107     }
00108   free (current->ptr);
00109   free (current);
00110 }
```

6.13.2 Variable Documentation

6.13.2.1 list_mutex

```
pthread_mutex_t list_mutex = PTHREAD_MUTEX_INITIALIZER
```

Definition at line 15 of file linked_list.c.

6.14 linked_list.c

Go to the documentation of this file.

```
00001 #include "linked_list.h"
00002 #include "config.h" // for ERROR_CODE, SUCCESS_CODE
00003
00004 #include <pthread.h> // for pthread_mutex_lock, pthread_mutex_unlock, PTHREAD_MUTEX_INITIALIZER
00005 #include <stdlib.h> // for malloc, free
00006
00007 struct Node
00008 {
00009   void *ptr;
00010   uint64_t size;
00011   struct Node **children;
```

```
00012   struct Node *next;
00013 };
00014
00015 pthread_mutex_t list_mutex = PTHREAD_MUTEX_INITIALIZER;
00016
00017 // create_node
00018 [[nodiscard]]
00019 struct Node *
00020 create_node (void *ptr, const uint64_t amount, void *const *children)
00021 {
00022   struct Node *new_node = (struct Node *)malloc (sizeof (struct Node));
00023   if (!new_node)
00024     {
00025       return NULL;
00026     }
00027   new_node->ptr = ptr;
00028   new_node->size = amount;
00029   new_node->children = NULL;
00030   if (amount <= 0)
00031     {
00032       new_node->next = NULL;
00033       return new_node;
00034     }
00035   new_node->children
00036       = (struct Node **)malloc (sizeof (struct Node *) * amount);
00037   if (!new_node->children)
00038     {
00039       free (new_node);
00040       return NULL;
00041     }
00042   for (uint64_t i = 0; i < amount; ++i)
00043     {
00044       new_node->children[i] = (struct Node *)malloc (sizeof (struct Node));
00045
00046       if (!new_node->children[i])
00047         {
00048           for (uint64_t j = 0; j < i; ++j)
00049             {
00050               free (new_node->children[j]);
00051             }
00052           free (new_node);
00053           return NULL;
00054         }
00055     }
00056   for (uint64_t i = 0; i < amount; ++i)
00057     {
00058       new_node->children[i]->size = 0;
00059       new_node->children[i]->next = NULL;
00060       new_node->children[i]->ptr = children[i];
00061       new_node->children[i]->children = NULL;
00062     }
00063   new_node->next = NULL;
00064   return new_node;
00065 }
00066
00067 // list_insert
00068 [[nodiscard]]
00069 int
00070 list_insert (struct Node **head, void *ptr, const uint64_t amount,
00071              void **children)
00072 {
00073   struct Node *new_node = create_node (ptr, amount, children);
00074   if (!new_node)
00075     {
00076       return ERROR_CODE;
00077     }
00078   pthread_mutex_lock (&list_mutex);
00079   if (!*head)
00080     {
00081       *head = new_node;
00082       pthread_mutex_unlock (&list_mutex);
00083       return SUCCESS_CODE;
00084     }
00085   struct Node *temp = *head;
00086   while (temp->next)
00087     {
00088       temp = temp->next;
00089     }
00090   temp->next = new_node;
00091   pthread_mutex_unlock (&list_mutex);
00092   return SUCCESS_CODE;
00093 }
00094
00095 // free_node
00096 void
00097 free_node (struct Node *current)
00098 {
```

```
00099   if (current->size > 0)
00100     {
00101       for (uint64_t i = 0; i < current->size; ++i)
00102         {
00103           free (current->children[i]->ptr);
00104           free (current->children[i]);
00105         }
00106       free (current->children);
00107     }
00108   free (current->ptr);
00109   free (current);
00110 }
00111
00112 // list_delete_node
00113 void
00114 list_delete_node (struct Node **head, const void *const ptr)
00115 {
00116   pthread_mutex_lock (&list_mutex);
00117   struct Node *current = *head;
00118   struct Node *prev = NULL;
00119
00120   if (current && current->ptr == ptr)
00121     {
00122       *head = current->next;
00123       free_node (current);
00124       pthread_mutex_unlock (&list_mutex);
00125       return;
00126     }
00127
00128   while (current && current->ptr != ptr)
00129     {
00130       prev = current;
00131       current = current->next;
00132     }
00133
00134   if (!current)
00135     {
00136       pthread_mutex_unlock (&list_mutex);
00137       return;
00138     }
00139
00140   prev->next = current->next;
00141   free_node (current);
00142   pthread_mutex_unlock (&list_mutex);
00143 }
00144
00145 // list_free
00146 void
00147 list_free (struct Node **head)
00148 {
00149   pthread_mutex_lock (&list_mutex);
00150   struct Node *current = *head;
00151   struct Node *next;
00152
00153   while (current)
00154     {
00155       next = current->next;
00156       free_node (current);
00157       current = next;
00158     }
00159
00160   *head = NULL;
00161   pthread_mutex_unlock (&list_mutex);
00162 }
```

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

```
#include "low_saurion.h"
#include "config.h"
#include "linked_list.h"
#include "threadpool.h"
#include <bits/types/struct_timeval.h>
#include <liburing.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <sys/eventfd.h>
```

Include dependency graph for low_saurion.c:

Classes

- struct request
- struct saurion_wrapper
- struct chunk_params

Macros

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

Functions

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

  Initializes a specified `iovec` structure with a message fragment.
- int allocate_iovec (struct iovec *iov, const uint64_t amount, const uint64_t pos, const uint64_t size, void **chd_ptr)
- int set_request (struct request **r, struct Node **l, uint64_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, const int client_socket, const int sel)
- static void add_efd (struct saurion *const s, const int client_socket, const 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, const int fd, const char *const str, const int sel)
- static void handle_accept (const struct saurion *const s, const int fd)
- static uint64_t calculate_max_iov_content (const struct request *req)
- static int handle_previous_message (struct chunk_params *p)
- static int handle_partial_message (struct chunk_params *p)
- static int handle_new_message (struct chunk_params *p)
- static int prepare_destination (struct chunk_params *p)
- static void copy_data (struct chunk_params *p, uint8_t *const ok)
- static uint8_t validate_and_update (struct chunk_params *const p, const uint8_t ok)
- static void read_chunk_free (struct chunk_params *const p)
- int read_chunk (void **dest, uint64_t *const len, struct request *const req)

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

  Creates a socket.

- struct saurion * saurion_create (uint32_t n_threads)

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

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

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

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

  Sends a message through a socket using io_uring.

Variables

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

6.15.1 Macro Definition Documentation

6.15.1.1 EV_ACC

```
#define EV_ACC 0
```

Definition at line 16 of file low_saurion.c.

6.15.1.2 EV_ERR

```
#define EV_ERR 4
```

Definition at line 20 of file low_saurion.c.

6.15.1.3 EV_REA

```
#define EV_REA 1
```

Definition at line 17 of file low_saurion.c.

6.15.1.4 EV_WAI

```
#define EV_WAI 3
```

Definition at line 19 of file low_saurion.c.

6.15.1.5 EV_WRI

```
#define EV_WRI 2
```

Definition at line 18 of file low_saurion.c.

6.15.1.6 MAX

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

Definition at line 36 of file low_saurion.c.

6.15.1.7 MIN

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

Definition at line 35 of file low_saurion.c.

6.15.2 Function Documentation

6.15.2.1 add_accept()

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

Definition at line 247 of file low_saurion.c.

```
00249 {
00250   int res = ERROR_CODE;
00251   pthread_mutex_lock (&s->m_rings[0]);
00252   while (res != SUCCESS_CODE)
00253     {
00254       struct io_uring_sqe *sqe = io_uring_get_sqe (&s->rings[0]);
00255       while (!sqe)
00256         {
00257           sqe = io_uring_get_sqe (&s->rings[0]);
00258           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00259         }
00260       struct request *req = NULL;
00261       if (!set_request (&req, &s->list, 0, NULL, 0))
00262         {
00263           free (sqe);
00264           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00265           res = ERROR_CODE;
00266           continue;
00267         }
00268       req->client_socket = 0;
00269       req->event_type = EV_ACC;
00270       io_uring_prep_accept (sqe, s->ss, (struct sockaddr *const)ca, cal, 0);
00271       io_uring_sqe_set_data (sqe, req);
00272       if (io_uring_submit (&s->rings[0]) < 0)
00273         {
00274           free (sqe);
00275           list_delete_node (&s->list, req);
00276           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00277           res = ERROR_CODE;
00278           continue;
00279         }
00280       res = SUCCESS_CODE;
00281     }
00282   pthread_mutex_unlock (&s->m_rings[0]);
00283 }
```

6.15.2.2 add_efd()

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

Definition at line 326 of file low_saurion.c.

```
00327 {
00328   add_fd (s, client_socket, sel);
00329 }
```

6.15.2.3 add_fd()

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

Definition at line 287 of file low_saurion.c.

```
00288 {
00289   int res = ERROR_CODE;
00290   pthread_mutex_lock (&s->m_rings[sel]);
00291   while (res != SUCCESS_CODE)
00292     {
00293       struct io_uring *ring = &s->rings[sel];
00294       struct io_uring_sqe *sqe = io_uring_get_sqe (ring);
00295       while (!sqe)
00296         {
00297           sqe = io_uring_get_sqe (ring);
00298           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00299         }
00300       struct request *req = NULL;
00301       if (!set_request (&req, &s->list, CHUNK_SZ, NULL, 0))
00302         {
00303           free (sqe);
00304           res = ERROR_CODE;
00305           continue;
00306         }
00307       req->event_type = EV_REA;
00308       req->client_socket = client_socket;
00309       io_uring_prep_readv (sqe, client_socket, &req->iov[0], req->iovec_count,
00310                            0);
00311       io_uring_sqe_set_data (sqe, req);
00312       if (io_uring_submit (ring) < 0)
00313         {
00314           free (sqe);
00315           list_delete_node (&s->list, req);
00316           res = ERROR_CODE;
00317           continue;
00318         }
00319       res = SUCCESS_CODE;
00320     }
00321   pthread_mutex_unlock (&s->m_rings[sel]);
00322 }
```

6.15.2.4 add_read()

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

Definition at line 333 of file low_saurion.c.

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

6.15.2.5 add_read_continue()

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

Definition at line 341 of file low_saurion.c.

```
00343 {
00344   pthread_mutex_lock (&s->m_rings[sel]);
00345   int res = ERROR_CODE;
00346   while (res != SUCCESS_CODE)
00347     {
00348       struct io_uring *ring = &s->rings[sel];
00349       struct io_uring_sqe *sqe = io_uring_get_sqe (ring);
00350       while (!sqe)
00351         {
00352           sqe = io_uring_get_sqe (ring);
00353           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
```

```
00354         }
00355       if (!set_request (&oreq, &s->list, oreq->prev_remain, NULL, 0))
00356         {
00357           free (sqe);
00358           res = ERROR_CODE;
00359           continue;
00360         }
00361       io_uring_prep_readv (sqe, oreq->client_socket, &oreq->iov[0],
00362                            oreq->iovec_count, 0);
00363       io_uring_sqe_set_data (sqe, oreq);
00364       if (io_uring_submit (ring) < 0)
00365         {
00366           free (sqe);
00367           list_delete_node (&s->list, oreq);
00368           res = ERROR_CODE;
00369           continue;
00370         }
00371       res = SUCCESS_CODE;
00372     }
00373   pthread_mutex_unlock (&s->m_rings[sel]);
00374 }
```

6.15.2.6 add_write()

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

Definition at line 378 of file low_saurion.c.

```
00380 {
00381   int res = ERROR_CODE;
00382   pthread_mutex_lock (&s->m_rings[sel]);
00383   while (res != SUCCESS_CODE)
00384     {
00385       struct io_uring *ring = &s->rings[sel];
00386       struct io_uring_sqe *sqe = io_uring_get_sqe (ring);
00387       while (!sqe)
00388         {
00389           sqe = io_uring_get_sqe (ring);
00390           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00391         }
00392       struct request *req = NULL;
00393       if (!set_request (&req, &s->list, strlen (str), (const void *const)str,
00394                         1))
00395         {
00396           free (sqe);
00397           res = ERROR_CODE;
00398           continue;
00399         }
00400       req->event_type = EV_WRI;
00401       req->client_socket = fd;
00402       io_uring_prep_writev (sqe, req->client_socket, req->iov,
00403                             req->iovec_count, 0);
00404       io_uring_sqe_set_data (sqe, req);
00405       if (io_uring_submit (ring) < 0)
00406         {
00407           free (sqe);
00408           list_delete_node (&s->list, req);
00409           res = ERROR_CODE;
00410           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00411           continue;
00412         }
00413       res = SUCCESS_CODE;
00414     }
00415   pthread_mutex_unlock (&s->m_rings[sel]);
00416 }
```

6.15.2.7 calculate_max_iov_content()

```
static uint64_t calculate_max_iov_content (
            const struct request * req )  [inline], [static]
```

Definition at line 432 of file low_saurion.c.

```
00433 {
00434   uint64_t max_iov_cont = 0;
00435   for (uint64_t i = 0; i < req->iovec_count; ++i)
00436     {
00437       max_iov_cont += req->iov[i].iov_len;
00438     }
00439   return max_iov_cont;
00440 }
```

6.15.2.8 copy_data()

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

Definition at line 574 of file low_saurion.c.

```
00575 {
00576   uint64_t curr_iov_msg_rem = 0;
00577   *ok = 1UL;
00578   while (1)
00579     {
00580       curr_iov_msg_rem = MIN (
00581           p->cont_rem, (p->req->iov[p->curr_iov].iov_len - p->curr_iov_off));
00582       memcpy ((uint8_t *)p->dest_ptr + p->dest_off,
00583               (uint8_t *)p->req->iov[p->curr_iov].iov_base + p->curr_iov_off,
00584               curr_iov_msg_rem);
00585       p->dest_off += curr_iov_msg_rem;
00586       p->curr_iov_off += curr_iov_msg_rem;
00587       p->cont_rem -= curr_iov_msg_rem;
00588
00589       if (p->cont_rem <= 0)
00590         {
00591           if (*(((uint8_t *)p->req->iov[p->curr_iov].iov_base)
00592                 + p->curr_iov_off)
00593               != 0)
00594             {
00595               *ok = 0UL;
00596             }
00597           *p->len = p->cont_sz;
00598           ++p->curr_iov_off;
00599           break;
00600         }
00601       if (p->curr_iov_off >= (p->req->iov[p->curr_iov].iov_len))
00602         {
00603           ++p->curr_iov;
00604           if (p->curr_iov == p->req->iovec_count)
00605             {
00606               break;
00607             }
00608           p->curr_iov_off = 0;
00609         }
00610     }
00611 }
```

6.15.2.9 handle_accept()

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

Definition at line 421 of file low_saurion.c.

```
00422 {
00423   if (s->cb.on_connected)
00424     {
00425       s->cb.on_connected (fd, s->cb.on_connected_arg);
00426     }
00427 }
```

6.15.2.10 handle_close()

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

Definition at line 775 of file low_saurion.c.

```
00776 {
00777   if (s->cb.on_closed)
00778     {
00779       s->cb.on_closed (req->client_socket, s->cb.on_closed_arg);
00780     }
00781   close (req->client_socket);
00782 }
```

6.15.2.11 handle_error()

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

Definition at line 763 of file low_saurion.c.

```
00764 {
00765   if (s->cb.on_error)
00766     {
00767       const char *resp = "ERROR";
00768       s->cb.on_error (req->client_socket, resp, (int64_t)strlen (resp),
00769                       s->cb.on_error_arg);
00770     }
00771 }
```

6.15.2.12 handle_event_read()

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

Definition at line 949 of file low_saurion.c.

```
00951 {
00952   if (cqe->res < 0)
00953     {
00954       handle_error (s, req);
00955     }
00956   if (cqe->res < 1)
00957     {
00958       handle_close (s, req);
00959     }
00960   if (cqe->res > 0)
00961     {
00962       handle_read (s, req);
00963     }
00964   list_delete_node (&s->list, req);
00965 }
```

6.15.2.13 handle_new_message()

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

Definition at line 522 of file low_saurion.c.

```
00523 {
00524   p->curr_iov = 0;
00525   p->curr_iov_off = 0;
00526
00527   p->cont_sz = *(uint64_t *)((uint8_t *)p->req->iov[p->curr_iov].iov_base
00528                              + p->curr_iov_off);
00529   p->cont_sz = ntohll (p->cont_sz);
00530   p->curr_iov_off += sizeof (uint64_t);
00531   p->cont_rem = p->cont_sz;
00532   p->dest_off = p->cont_sz - p->cont_rem;
00533
00534   if (p->cont_rem <= p->max_iov_cont)
00535     {
00536       *p->dest = malloc (p->cont_sz);
00537       if (!*p->dest)
00538         {
00539           return ERROR_CODE; // Error al asignar memoria.
00540         }
00541       p->dest_ptr = *p->dest;
00542     }
00543   else
00544     {
00545       p->req->prev = malloc (p->cont_sz);
00546       if (!p->req->prev)
00547         {
00548           return ERROR_CODE; // Error al asignar memoria.
00549         }
00550       p->dest_ptr = p->req->prev;
00551       *p->dest = NULL;
00552     }
00553   return SUCCESS_CODE;
00554 }
```

6.15.2.14 handle_partial_message()

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

Definition at line 484 of file low_saurion.c.

```
00485 {
00486   p->curr_iov = p->req->next_iov;
00487   p->curr_iov_off = p->req->next_offset;
00488
00489   p->cont_sz = *(uint64_t *)((uint8_t *)p->req->iov[p->curr_iov].iov_base
00490                              + p->curr_iov_off);
00491   p->cont_sz = ntohll (p->cont_sz);
00492   p->curr_iov_off += sizeof (uint64_t);
00493   p->cont_rem = p->cont_sz;
00494   p->dest_off = p->cont_sz - p->cont_rem;
00495
00496   if ((p->curr_iov_off + p->cont_rem + 1) <= p->max_iov_cont)
00497     {
00498       *p->dest = malloc (p->cont_sz);
00499       if (!*p->dest)
00500         {
00501           return ERROR_CODE;
00502         }
00503       p->dest_ptr = *p->dest;
00504     }
00505   else
00506     {
00507       p->req->prev = malloc (p->cont_sz);
00508       if (!p->req->prev)
00509         {
00510           return ERROR_CODE;
00511         }
00512       p->dest_ptr = p->req->prev;
00513       *p->dest = NULL;
00514       *p->len = 0;
00515     }
00516   return SUCCESS_CODE;
00517 }
```

6.15.2.15 handle_previous_message()

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

Definition at line 459 of file low_saurion.c.

```
00460 {
00461   p->cont_sz = p->req->prev_size;
00462   p->cont_rem = p->req->prev_remain;
00463   p->dest_off = p->cont_sz - p->cont_rem;
00464
00465   if (p->cont_rem <= p->max_iov_cont)
00466     {
00467       *p->dest = p->req->prev;
00468       p->dest_ptr = *p->dest;
00469       p->req->prev = NULL;
00470       p->req->prev_size = 0;
00471       p->req->prev_remain = 0;
00472     }
00473   else
00474     {
00475       p->dest_ptr = p->req->prev;
00476       *p->dest = NULL;
00477     }
00478   return SUCCESS_CODE;
00479 }
```

6.15.2.16 handle_read()

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

Definition at line 714 of file low_saurion.c.

```
00715 {
00716   void *msg = NULL;
00717   uint64_t len = 0;
00718   while (1)
00719     {
00720       if (!read_chunk (&msg, &len, req))
00721         {
00722           break;
00723         }
00724       if (req->next_iov || req->next_offset)
00725         {
00726           if (s->cb.on_readed && msg)
00727             {
00728               s->cb.on_readed (req->client_socket, msg, len,
00729                                s->cb.on_readed_arg);
00730             }
00731           free (msg);
00732           msg = NULL;
00733           continue;
00734         }
00735       if (req->prev && req->prev_size && req->prev_remain)
00736         {
00737           add_read_continue (s, req, next (s));
00738           return;
00739         }
00740       if (s->cb.on_readed && msg)
00741         {
00742           s->cb.on_readed (req->client_socket, msg, len, s->cb.on_readed_arg);
00743         }
00744       free (msg);
00745       msg = NULL;
00746       break;
00747     }
00748   add_read (s, req->client_socket);
00749 }
```

6.15.2.17 handle_write()

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

Definition at line 753 of file low_saurion.c.

```
00754 {
00755   if (s->cb.on_wrote)
00756     {
00757       s->cb.on_wrote (fd, s->cb.on_wrote_arg);
00758     }
00759 }
```

6.15.2.18 htonll()

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

Definition at line 56 of file low_saurion.c.

```
00057 {
00058   int num = 42;
```

```
00059   if (*(char *)&num == 42)
00060     {
00061       uint32_t high_part = htonl ((uint32_t)(value » 32));
00062       uint32_t low_part = htonl ((uint32_t)(value & 0xFFFFFFFFLL));
00063       return ((uint64_t)low_part « 32) | high_part;
00064     }
00065   return value;
00066 }
```

6.15.2.19 next()

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

Definition at line 48 of file low_saurion.c.

```
00049 {
00050   s->next = (s->next + 1) % s->n_threads;
00051   return s->next;
00052 }
```

6.15.2.20 ntohll()

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

Definition at line 70 of file low_saurion.c.

```
00071 {
00072   int num = 42;
00073   if (*(char *)&num == 42)
00074     {
00075       uint32_t high_part = ntohl ((uint32_t)(value » 32));
00076       uint32_t low_part = ntohl ((uint32_t)(value & 0xFFFFFFFFLL));
00077       return ((uint64_t)low_part « 32) | high_part;
00078     }
00079   return value;
00080 }
```

6.15.2.21 prepare_destination()

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

Definition at line 559 of file low_saurion.c.

```
00560 {
00561   if (p->req->prev && p->req->prev_size && p->req->prev_remain)
00562     {
00563       return handle_previous_message (p);
00564     }
00565   if (p->req->next_iov || p->req->next_offset)
00566     {
00567       return handle_partial_message (p);
00568     }
00569   return handle_new_message (p);
00570 }
```

6.15.2.22 read_chunk_free()

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

Definition at line 652 of file low_saurion.c.

```
00653 {
00654   free (p->dest_ptr);
00655   p->dest_ptr = NULL;
00656   *p->dest = NULL;
00657   *p->len = 0;
00658   p->req->next_iov = 0;
00659   p->req->next_offset = 0;
00660   for (uint64_t i = p->curr_iov; i < p->req->iovec_count; ++i)
00661     {
00662       for (uint64_t j = p->curr_iov_off; j < p->req->iov[i].iov_len; ++j)
00663         {
00664           uint8_t foot = *((uint8_t *)p->req->iov[i].iov_base) + j;
00665           if (foot == 0)
00666             {
00667               p->req->next_iov = i;
00668               p->req->next_offset = (j + 1) % p->req->iov[i].iov_len;
00669               return;
00670             }
00671         }
00672     }
00673 }
```

6.15.2.23 saurion_worker_master()

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

Definition at line 1027 of file low_saurion.c.

```
01028 {
01029   LOG_INIT (" ");
01030   struct saurion *const s = (struct saurion *const)arg;
01031   struct sockaddr_in client_addr;
01032   socklen_t client_addr_len = sizeof (client_addr);
01033
01034   add_efd (s, s->efds[0], 0);
01035   add_accept (s, &client_addr, &client_addr_len);
01036
01037   pthread_mutex_lock (&s->status_m);
01038   ++s->status;
01039   pthread_cond_broadcast (&s->status_c);
01040   pthread_mutex_unlock (&s->status_m);
01041   while (1)
01042     {
01043       int ret
01044           = saurion_worker_master_loop_it (s, &client_addr, &client_addr_len);
01045       if (ret == ERROR_CODE || ret == CRITICAL_CODE)
01046         {
01047           break;
01048         }
01049     }
01050   pthread_mutex_lock (&s->status_m);
01051   --s->status;
01052   pthread_cond_signal (&s->status_c);
01053   pthread_mutex_unlock (&s->status_m);
01054   LOG_END (" ");
01055   return;
01056 }
```

6.15.2.24 saurion_worker_master_loop_it()

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

Definition at line 970 of file low_saurion.c.

```
00973 {
00974   LOG_INIT (" ");
00975   struct io_uring ring = s->rings[0];
00976   struct io_uring_cqe *cqe = NULL;
00977   int ret = io_uring_wait_cqe (&ring, &cqe);
00978   if (ret < 0)
00979     {
00980       free (cqe);
00981       LOG_END (" ");
00982       return CRITICAL_CODE;
00983     }
00984   struct request *req = (struct request *)cqe->user_data;
00985   if (!req)
00986     {
00987       io_uring_cqe_seen (&s->rings[0], cqe);
00988       LOG_END (" ");
00989       return SUCCESS_CODE;
00990     }
00991   if (cqe->res < 0)
00992     {
00993       list_delete_node (&s->list, req);
00994       LOG_END (" ");
00995       return CRITICAL_CODE;
00996     }
00997   if (req->client_socket == s->efds[0])
00998     {
00999       io_uring_cqe_seen (&s->rings[0], cqe);
01000       list_delete_node (&s->list, req);
01001       LOG_END (" ");
01002       return ERROR_CODE;
01003     }
01004   io_uring_cqe_seen (&s->rings[0], cqe);
01005   switch (req->event_type)
01006     {
01007     case EV_ACC:
01008       handle_accept (s, cqe->res);
01009       add_accept (s, client_addr, client_addr_len);
01010       add_read (s, cqe->res);
01011       list_delete_node (&s->list, req);
01012       break;
01013     case EV_REA:
01014       handle_event_read (cqe, s, req);
01015       break;
01016     case EV_WRI:
01017       handle_write (s, req->client_socket);
01018       list_delete_node (&s->list, req);
01019       break;
01020     }
01021   LOG_END (" ");
01022   return SUCCESS_CODE;
01023 }
```

6.15.2.25 saurion_worker_slave()

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

Definition at line 1112 of file low_saurion.c.

```
01113 {
01114   LOG_INIT (" ");
01115   struct saurion_wrapper *const ss = (struct saurion_wrapper *const)arg;
01116   struct saurion *s = ss->s;
01117   const int sel = ss->sel;
01118   free (ss);
01119
01120   add_efd (s, s->efds[sel], sel);
```

```
01121
01122   pthread_mutex_lock (&s->status_m);
01123   ++s->status;
01124   pthread_cond_broadcast (&s->status_c);
01125   pthread_mutex_unlock (&s->status_m);
01126   while (1)
01127     {
01128       int res = saurion_worker_slave_loop_it (s, sel);
01129       if (res == ERROR_CODE || res == CRITICAL_CODE)
01130         {
01131           break;
01132         }
01133     }
01134   pthread_mutex_lock (&s->status_m);
01135   --s->status;
01136   pthread_cond_signal (&s->status_c);
01137   pthread_mutex_unlock (&s->status_m);
01138   LOG_END (" ");
01139   return;
01140 }
```

6.15.2.26 saurion_worker_slave_loop_it()

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

Definition at line 1061 of file low_saurion.c.

```
01062 {
01063   LOG_INIT (" ");
01064   struct io_uring ring = s->rings[sel];
01065   struct io_uring_cqe *cqe = NULL;
01066
01067   add_efd (s, s->efds[sel], sel);
01068   int ret = io_uring_wait_cqe (&ring, &cqe);
01069   if (ret < 0)
01070     {
01071       free (cqe);
01072       LOG_END (" ");
01073       return CRITICAL_CODE;
01074     }
01075   struct request *req = (struct request *)cqe->user_data;
01076   if (!req)
01077     {
01078       io_uring_cqe_seen (&ring, cqe);
01079       LOG_END (" ");
01080       return SUCCESS_CODE;
01081     }
01082   if (cqe->res < 0)
01083     {
01084       list_delete_node (&s->list, req);
01085       LOG_END (" ");
01086       return CRITICAL_CODE;
01087     }
01088   if (req->client_socket == s->efds[sel])
01089     {
01090       io_uring_cqe_seen (&ring, cqe);
01091       list_delete_node (&s->list, req);
01092       LOG_END (" ");
01093       return ERROR_CODE;
01094     }
01095   io_uring_cqe_seen (&ring, cqe);
01096   switch (req->event_type)
01097     {
01098     case EV_REA:
01099       handle_event_read (cqe, s, req);
01100       break;
01101     case EV_WRI:
01102       handle_write (s, req->client_socket);
01103       list_delete_node (&s->list, req);
01104       break;
01105     }
01106   LOG_END (" ");
01107   return SUCCESS_CODE;
01108 }
```

6.15.2.27 validate_and_update()

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

Definition at line 616 of file low_saurion.c.

```
00617 {
00618   if (p->req->prev)
00619     {
00620       p->req->prev_size = p->cont_sz;
00621       p->req->prev_remain = p->cont_rem;
00622       *p->dest = NULL;
00623       *p->len = 0;
00624     }
00625   else
00626     {
00627       p->req->prev_size = 0;
00628       p->req->prev_remain = 0;
00629     }
00630   if (p->curr_iov < p->req->iovec_count)
00631     {
00632       uint64_t next_sz
00633           = *(uint64_t *)(((uint8_t *)p->req->iov[p->curr_iov].iov_base)
00634                           + p->curr_iov_off);
00635       if ((p->req->iov[p->curr_iov].iov_len > p->curr_iov_off) && next_sz)
00636         {
00637           p->req->next_iov = p->curr_iov;
00638           p->req->next_offset = p->curr_iov_off;
00639         }
00640       else
00641         {
00642           p->req->next_iov = 0;
00643           p->req->next_offset = 0;
00644         }
00645     }
00646
00647   return ok ?  SUCCESS_CODE : ERROR_CODE;
00648 }
```

6.15.3 Variable Documentation

6.15.3.1 TIMEOUT_RETRY_SPEC

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

Definition at line 38 of file low_saurion.c.

6.16 low_saurion.c

Go to the documentation of this file.

```
00001 #include "low_saurion.h"
00002 #include "config.h"      // for ERROR_CODE, SUCCESS_CODE, CHUNK_SZ
00003 #include "linked_list.h" // for list_delete_node, list_free, list_insert
00004 #include "threadpool.h"  // for threadpool_add, threadpool_create
00005
00006 #include <bits/types/struct_timeval.h> // for struct timeval
00007 #include <liburing.h>     // for io_uring_get_sqe, io_uring, io_uring_...
00008 #include <netinet/in.h>  // for sockaddr_in, INADDR_ANY, in_addr
00009 #include <stdlib.h>      // for free, malloc
00010 #include <string.h>      // for memset, memcpy, strlen
00011 #include <sys/eventfd.h> // for eventfd, EFD_NONBLOCK
00012
00013 struct Node;
00014 struct iovec;
```

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

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

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

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

```
00363       io_uring_sqe_set_data (sqe, oreq);
00364       if (io_uring_submit (ring) < 0)
00365         {
00366           free (sqe);
00367           list_delete_node (&s->list, oreq);
00368           res = ERROR_CODE;
00369           continue;
00370         }
00371       res = SUCCESS_CODE;
00372     }
00373   pthread_mutex_unlock (&s->m_rings[sel]);
00374 }
00375
00376 // add_write
00377 static inline void
00378 add_write (struct saurion *const s, const int fd, const char *const str,
00379            const int sel)
00380 {
00381   int res = ERROR_CODE;
00382   pthread_mutex_lock (&s->m_rings[sel]);
00383   while (res != SUCCESS_CODE)
00384     {
00385       struct io_uring *ring = &s->rings[sel];
00386       struct io_uring_sqe *sqe = io_uring_get_sqe (ring);
00387       while (!sqe)
00388         {
00389           sqe = io_uring_get_sqe (ring);
00390           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00391         }
00392       struct request *req = NULL;
00393       if (!set_request (&req, &s->list, strlen (str), (const void *const)str,
00394                         1))
00395         {
00396           free (sqe);
00397           res = ERROR_CODE;
00398           continue;
00399         }
00400       req->event_type = EV_WRI;
00401       req->client_socket = fd;
00402       io_uring_prep_writev (sqe, req->client_socket, req->iov,
00403                             req->iovec_count, 0);
00404       io_uring_sqe_set_data (sqe, req);
00405       if (io_uring_submit (ring) < 0)
00406         {
00407           free (sqe);
00408           list_delete_node (&s->list, req);
00409           res = ERROR_CODE;
00410           nanosleep (&TIMEOUT_RETRY_SPEC, NULL);
00411           continue;
00412         }
00413       res = SUCCESS_CODE;
00414     }
00415   pthread_mutex_unlock (&s->m_rings[sel]);
00416 }
00417
00418 /****************** HANDLERS ******************/
00419 // handle_accept
00420 static inline void
00421 handle_accept (const struct saurion *const s, const int fd)
00422 {
00423   if (s->cb.on_connected)
00424     {
00425       s->cb.on_connected (fd, s->cb.on_connected_arg);
00426     }
00427 }
00428
00429 // calculate_max_iov_content
00430 [[nodiscard]]
00431 static inline uint64_t
00432 calculate_max_iov_content (const struct request *req)
00433 {
00434   uint64_t max_iov_cont = 0;
00435   for (uint64_t i = 0; i < req->iovec_count; ++i)
00436     {
00437       max_iov_cont += req->iov[i].iov_len;
00438     }
00439   return max_iov_cont;
00440 }
00441
00442 struct chunk_params
00443 {
00444   void **dest;
00445   void *dest_ptr;
00446   uint64_t dest_off;
00447   struct request *req;
00448   uint64_t cont_sz;
00449   uint64_t cont_rem;
```

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

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

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

```
00711
00712 // handle_read
00713 static inline void
00714 handle_read (struct saurion *const s, struct request *const req)
00715 {
00716   void *msg = NULL;
00717   uint64_t len = 0;
00718   while (1)
00719     {
00720       if (!read_chunk (&msg, &len, req))
00721         {
00722           break;
00723         }
00724       if (req->next_iov || req->next_offset)
00725         {
00726           if (s->cb.on_readed && msg)
00727             {
00728               s->cb.on_readed (req->client_socket, msg, len,
00729                                s->cb.on_readed_arg);
00730             }
00731           free (msg);
00732           msg = NULL;
00733           continue;
00734         }
00735       if (req->prev && req->prev_size && req->prev_remain)
00736         {
00737           add_read_continue (s, req, next (s));
00738           return;
00739         }
00740       if (s->cb.on_readed && msg)
00741         {
00742           s->cb.on_readed (req->client_socket, msg, len, s->cb.on_readed_arg);
00743         }
00744       free (msg);
00745       msg = NULL;
00746       break;
00747     }
00748   add_read (s, req->client_socket);
00749 }
00750
00751 // handle_write
00752 static inline void
00753 handle_write (const struct saurion *const s, const int fd)
00754 {
00755   if (s->cb.on_wrote)
00756     {
00757       s->cb.on_wrote (fd, s->cb.on_wrote_arg);
00758     }
00759 }
00760
00761 // handle_error
00762 static inline void
00763 handle_error (const struct saurion *const s, const struct request *const req)
00764 {
00765   if (s->cb.on_error)
00766     {
00767       const char *resp = "ERROR";
00768       s->cb.on_error (req->client_socket, resp, (int64_t)strlen (resp),
00769                       s->cb.on_error_arg);
00770     }
00771 }
00772
00773 // handle_close
00774 static inline void
00775 handle_close (const struct saurion *const s, const struct request *const req)
00776 {
00777   if (s->cb.on_closed)
00778     {
00779       s->cb.on_closed (req->client_socket, s->cb.on_closed_arg);
00780     }
00781   close (req->client_socket);
00782 }
00783
00784 /****************** INTERFACE ******************/
00785 // saurion_set_socket
00786 [[nodiscard]] int
00787 saurion_set_socket (const int p)
00788 {
00789   int sock = 0;
00790   struct sockaddr_in srv_addr;
00791
00792   sock = socket (PF_INET, SOCK_STREAM, 0);
00793   if (sock < 1)
00794     {
00795       return ERROR_CODE;
00796     }
00797
```

```
00798   int enable = 1;
00799   if (setsockopt (sock, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof (int)) < 0)
00800     {
00801       return ERROR_CODE;
00802     }
00803   struct timeval t_out;
00804   t_out.tv_sec = TIMEOUT_IDLE / 1000L;
00805   t_out.tv_usec = TIMEOUT_IDLE % 1000L;
00806   if (setsockopt (sock, SOL_SOCKET, SO_RCVTIMEO, &t_out, sizeof (t_out)) < 0)
00807     {
00808       return ERROR_CODE;
00809     }
00810
00811   memset (&srv_addr, 0, sizeof (srv_addr));
00812   srv_addr.sin_family = AF_INET;
00813   srv_addr.sin_port = htons (p);
00814   srv_addr.sin_addr.s_addr = htonl (INADDR_ANY);
00815
00816   if (bind (sock, (const struct sockaddr *)&srv_addr, sizeof (srv_addr)) < 0)
00817     {
00818       return ERROR_CODE;
00819     }
00820
00821   constexpr int num_queue = (ACCEPT_QUEUE > 0 ?  ACCEPT_QUEUE : SOMAXCONN);
00822   if (listen (sock, num_queue) < 0)
00823     {
00824       return ERROR_CODE;
00825     }
00826
00827   return sock;
00828 }
00829
00830 // saurion_create
00831 [[nodiscard]]
00832 struct saurion *
00833 saurion_create (uint32_t n_threads)
00834 {
00835   LOG_INIT (" ");
00836   struct saurion *p = (struct saurion *)malloc (sizeof (struct saurion));
00837   if (!p)
00838     {
00839       LOG_END (" ");
00840       return NULL;
00841     }
00842   int ret = 0;
00843   ret = pthread_mutex_init (&p->status_m, NULL);
00844   if (ret)
00845     {
00846       free (p);
00847       LOG_END (" ");
00848       return NULL;
00849     }
00850   ret = pthread_cond_init (&p->status_c, NULL);
00851   if (ret)
00852     {
00853       free (p);
00854       LOG_END (" ");
00855       return NULL;
00856     }
00857   p->m_rings
00858       = (pthread_mutex_t *)malloc (n_threads * sizeof (pthread_mutex_t));
00859   if (!p->m_rings)
00860     {
00861       free (p);
00862       LOG_END (" ");
00863       return NULL;
00864     }
00865   for (uint32_t i = 0; i < n_threads; ++i)
00866     {
00867       pthread_mutex_init (&(p->m_rings[i]), NULL);
00868     }
00869   p->ss = 0;
00870   n_threads = (n_threads < 2 ?  2 :  n_threads);
00871   n_threads = (n_threads > NUM_CORES ? NUM_CORES : n_threads);
00872   p->n_threads = n_threads;
00873   p->status = 0;
00874   p->list = NULL;
00875   p->cb.on_connected = NULL;
00876   p->cb.on_connected_arg = NULL;
00877   p->cb.on_readed = NULL;
00878   p->cb.on_readed_arg = NULL;
00879   p->cb.on_wrote = NULL;
00880   p->cb.on_wrote_arg = NULL;
00881   p->cb.on_closed = NULL;
00882   p->cb.on_closed_arg = NULL;
00883   p->cb.on_error = NULL;
00884   p->cb.on_error_arg = NULL;
```

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

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

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

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

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

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

Include dependency graph for main.c:

6.18 main.c

Go to the documentation of this file.

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

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

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

```
#include <stdexcept>
#include <unistd.h>
```

Include dependency graph for saurion.cpp:

6.20 saurion.cpp

Go to the documentation of this file.

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

```
00074   s->cb.on_error_arg = arg;
00075   return this;
00076 }
00077
00078 void
00079 Saurion::send (const int fd, const char *const msg) noexcept
00080 {
00081   saurion_send (this->s, fd, msg);
00082 }
```

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

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

Include dependency graph for threadpool.c:

Classes

- struct task
- struct threadpool

  Represents a thread pool.

Macros

- #define TRUE 1
- #define FALSE 0

Functions

- struct threadpool * threadpool_create (uint64_t num_threads)

  Creates a new thread pool with the specified number of threads.
- struct threadpool * threadpool_create_default (void)

  Creates a new thread pool with the default number of threads (equal to the number of CPU cores).
- void * threadpool_worker (void *arg)
- void threadpool_init (struct threadpool *pool)

  Initializes the thread pool, starting the worker threads.
- void threadpool_add (struct threadpool *pool, void(*function)(void *), void *argument)

  Adds a task to the thread pool.
- void threadpool_stop (struct threadpool *pool)

  Stops all threads in the thread pool and prevents further tasks from being added.
- int threadpool_empty (struct threadpool *pool)

  Checks if the thread pool's task queue is empty.
- void threadpool_wait_empty (struct threadpool *pool)

  Waits until the task queue becomes empty.
- void threadpool_destroy (struct threadpool *pool)

  Destroys the thread pool, freeing all allocated resources.

6.21.1 Macro Definition Documentation

6.21.1.1 FALSE

```
#define FALSE 0
```

Definition at line 7 of file threadpool.c.

6.21.1.2 TRUE

```
#define TRUE 1
```

Definition at line 6 of file threadpool.c.

6.21.2 Function Documentation

6.21.2.1 threadpool_worker()

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

Definition at line 100 of file threadpool.c.

```
00101 {
00102   LOG_INIT (" ");
00103   struct threadpool *pool = (struct threadpool *)arg;
00104   while (TRUE)
00105     {
00106       pthread_mutex_lock (&pool->queue_lock);
00107       while (pool->task_queue_head == NULL && !pool->stop)
00108         {
00109           pthread_cond_wait (&pool->queue_cond, &pool->queue_lock);
00110         }
00111
00112       if (pool->stop && pool->task_queue_head == NULL)
00113         {
00114           pthread_mutex_unlock (&pool->queue_lock);
00115           break;
00116         }
00117
00118       struct task *task = pool->task_queue_head;
00119       if (task != NULL)
00120         {
00121           pool->task_queue_head = task->next;
00122           if (pool->task_queue_head == NULL)
00123             pool->task_queue_tail = NULL;
00124
00125           if (pool->task_queue_head == NULL)
00126             {
00127               pthread_cond_signal (&pool->empty_cond);
00128             }
00129         }
00130       pthread_mutex_unlock (&pool->queue_lock);
00131
00132       if (task != NULL)
00133         {
00134           task->function (task->argument);
00135           free (task);
00136         }
00137     }
00138   LOG_END (" ");
00139   pthread_exit (NULL);
00140   return NULL;
00141 }
```

6.22 threadpool.c

Go to the documentation of this file.

```
00001 #include "threadpool.h"
00002 #include "config.h"
00003 #include <pthread.h> // for pthread_mutex_unlock, pthread_mutex_lock
00004 #include <stdlib.h>  // for free, malloc
00005
00006 #define TRUE 1
00007 #define FALSE 0
00008
00009 struct task
00010 {
00011   void (*function) (void *);
00012   void *argument;
00013   struct task *next;
00014 };
00015
00016 struct threadpool
00017 {
00018   pthread_t *threads;
00019   uint64_t num_threads;
00020   struct task *task_queue_head;
00021   struct task *task_queue_tail;
00022   pthread_mutex_t queue_lock;
00023   pthread_cond_t queue_cond;
00024   pthread_cond_t empty_cond;
00025   int stop;
00026   int started;
00027 };
00028
00029 struct threadpool *
00030 threadpool_create (uint64_t num_threads)
00031 {
00032   LOG_INIT (" ");
00033   struct threadpool *pool = malloc (sizeof (struct threadpool));
00034   if (pool == NULL)
00035     {
00036       LOG_END (" ");
00037       return NULL;
00038     }
00039   if (num_threads < 3)
00040     {
00041       num_threads = 3;
00042     }
00043   if (num_threads > NUM_CORES)
00044     {
00045       num_threads = NUM_CORES;
00046     }
00047
00048   pool->num_threads = num_threads;
00049   pool->threads = malloc (sizeof (pthread_t) * num_threads);
00050   if (pool->threads == NULL)
00051     {
00052       free (pool);
00053       LOG_END (" ");
00054       return NULL;
00055     }
00056
00057   pool->task_queue_head = NULL;
00058   pool->task_queue_tail = NULL;
00059   pool->stop = FALSE;
00060   pool->started = FALSE;
00061
00062   if (pthread_mutex_init (&pool->queue_lock, NULL) != 0)
00063     {
00064       free (pool->threads);
00065       free (pool);
00066       LOG_END (" ");
00067       return NULL;
00068     }
00069
00070   if (pthread_cond_init (&pool->queue_cond, NULL) != 0)
00071     {
00072       pthread_mutex_destroy (&pool->queue_lock);
00073       free (pool->threads);
00074       free (pool);
00075       LOG_END (" ");
00076       return NULL;
00077     }
00078
00079   if (pthread_cond_init (&pool->empty_cond, NULL) != 0)
00080     {
00081       pthread_mutex_destroy (&pool->queue_lock);
00082       pthread_cond_destroy (&pool->queue_cond);
```

```
00083       free (pool->threads);
00084       free (pool);
00085       LOG_END (" ");
00086       return NULL;
00087     }
00088
00089   LOG_END (" ");
00090   return pool;
00091 }
00092
00093 struct threadpool *
00094 threadpool_create_default (void)
00095 {
00096   return threadpool_create (NUM_CORES);
00097 }
00098
00099 void *
00100 threadpool_worker (void *arg)
00101 {
00102   LOG_INIT (" ");
00103   struct threadpool *pool = (struct threadpool *)arg;
00104   while (TRUE)
00105     {
00106       pthread_mutex_lock (&pool->queue_lock);
00107       while (pool->task_queue_head == NULL && !pool->stop)
00108         {
00109           pthread_cond_wait (&pool->queue_cond, &pool->queue_lock);
00110         }
00111
00112       if (pool->stop && pool->task_queue_head == NULL)
00113         {
00114           pthread_mutex_unlock (&pool->queue_lock);
00115           break;
00116         }
00117
00118       struct task *task = pool->task_queue_head;
00119       if (task != NULL)
00120         {
00121           pool->task_queue_head = task->next;
00122           if (pool->task_queue_head == NULL)
00123             pool->task_queue_tail = NULL;
00124
00125           if (pool->task_queue_head == NULL)
00126             {
00127               pthread_cond_signal (&pool->empty_cond);
00128             }
00129         }
00130       pthread_mutex_unlock (&pool->queue_lock);
00131
00132       if (task != NULL)
00133         {
00134           task->function (task->argument);
00135           free (task);
00136         }
00137     }
00138   LOG_END (" ");
00139   pthread_exit (NULL);
00140   return NULL;
00141 }
00142
00143 void
00144 threadpool_init (struct threadpool *pool)
00145 {
00146   LOG_INIT (" ");
00147   if (pool == NULL || pool->started)
00148     {
00149       LOG_END (" ");
00150       return;
00151     }
00152   for (uint64_t i = 0; i < pool->num_threads; i++)
00153     {
00154       if (pthread_create (&pool->threads[i], NULL, threadpool_worker,
00155                           (void *)pool)
00156           != 0)
00157         {
00158           pool->stop = TRUE;
00159           break;
00160         }
00161     }
00162   pool->started = TRUE;
00163   LOG_END (" ");
00164 }
00165
00166 void
00167 threadpool_add (struct threadpool *pool, void (*function) (void *),
00168                 void *argument)
00169 {
```

```
00170   LOG_INIT (" ");
00171   if (pool == NULL || function == NULL)
00172     {
00173       LOG_END (" ");
00174       return;
00175     }
00176
00177   struct task *new_task = malloc (sizeof (struct task));
00178   if (new_task == NULL)
00179     {
00180       LOG_END (" ");
00181       return;
00182     }
00183
00184   new_task->function = function;
00185   new_task->argument = argument;
00186   new_task->next = NULL;
00187
00188   pthread_mutex_lock (&pool->queue_lock);
00189
00190   if (pool->task_queue_head == NULL)
00191     {
00192       pool->task_queue_head = new_task;
00193       pool->task_queue_tail = new_task;
00194     }
00195   else
00196     {
00197       pool->task_queue_tail->next = new_task;
00198       pool->task_queue_tail = new_task;
00199     }
00200   pthread_cond_signal (&pool->queue_cond);
00201
00202   pthread_mutex_unlock (&pool->queue_lock);
00203   LOG_END (" ");
00204 }
00205
00206 void
00207 threadpool_stop (struct threadpool *pool)
00208 {
00209   LOG_INIT (" ");
00210   if (pool == NULL || !pool->started)
00211     {
00212       LOG_END (" ");
00213       return;
00214     }
00215   threadpool_wait_empty (pool);
00216
00217   pthread_mutex_lock (&pool->queue_lock);
00218   pool->stop = TRUE;
00219   pthread_cond_broadcast (&pool->queue_cond);
00220   pthread_mutex_unlock (&pool->queue_lock);
00221
00222   for (uint64_t i = 0; i < pool->num_threads; i++)
00223     {
00224       pthread_join (pool->threads[i], NULL);
00225     }
00226   pool->started = FALSE;
00227   LOG_END (" ");
00228 }
00229
00230 int
00231 threadpool_empty (struct threadpool *pool)
00232 {
00233   LOG_INIT (" ");
00234   if (pool == NULL)
00235     {
00236       LOG_END (" ");
00237       return TRUE;
00238     }
00239   pthread_mutex_lock (&pool->queue_lock);
00240   int empty = (pool->task_queue_head == NULL);
00241   pthread_mutex_unlock (&pool->queue_lock);
00242   LOG_END (" ");
00243   return empty;
00244 }
00245
00246 void
00247 threadpool_wait_empty (struct threadpool *pool)
00248 {
00249   LOG_INIT (" ");
00250   if (pool == NULL)
00251     {
00252       LOG_END (" ");
00253       return;
00254     }
00255   pthread_mutex_lock (&pool->queue_lock);
00256   while (pool->task_queue_head != NULL)
```

```
00257     {
00258       pthread_cond_wait (&pool->empty_cond, &pool->queue_lock);
00259     }
00260   pthread_mutex_unlock (&pool->queue_lock);
00261   LOG_END (" ");
00262 }
00263
00264 void
00265 threadpool_destroy (struct threadpool *pool)
00266 {
00267   LOG_INIT (" ");
00268   if (pool == NULL)
00269     {
00270       LOG_END (" ");
00271       return;
00272     }
00273   threadpool_stop (pool);
00274
00275   pthread_mutex_destroy (&pool->queue_lock);
00276   pthread_cond_destroy (&pool->queue_cond);
00277   pthread_cond_destroy (&pool->empty_cond);
00278
00279   free (pool->threads);
00280   free (pool);
00281   LOG_END (" ");
00282 }
```

Index