

Mini Memcached Distribuído

Generated by Doxygen 1.9.8

1 Class Index	1
1.1 Class List	1
2 File Index	3
2.1 File List	3
3 Class Documentation	5
3.1 Args Struct Reference	5
3.1.1 Detailed Description	5
3.1.2 Member Data Documentation	5
3.1.2.1 memory_limit	5
3.1.2.2 num_threads	5
3.1.2.3 port	6
3.2 AtomCounter Struct Reference	6
3.2.1 Detailed Description	6
3.2.2 Member Data Documentation	6
3.2.2.1 counter	6
3.2.2.2 lock	6
3.3 Cache Struct Reference	6
3.3.1 Detailed Description	7
3.3.2 Member Data Documentation	7
3.3.2.1 buckets	7
3.3.2.2 hash_function	7
3.3.2.3 num_buckets	7
3.3.2.4 num_zones	7
3.3.2.5 queue	7
3.3.2.6 stats	7
3.3.2.7 zone_locks	8
3.4 CacheStats Struct Reference	8
3.4.1 Detailed Description	8
3.4.2 Member Data Documentation	8
3.4.2.1 allocated_memory	8
3.4.2.2 del_counter	8
3.4.2.3 evict_counter	8
3.4.2.4 get_counter	9
3.4.2.5 key_counter	9
3.4.2.6 put_counter	9
3.5 ClientData Struct Reference	9
3.5.1 Detailed Description	9
3.5.2 Member Data Documentation	10
3.5.2.1 cleaning	10
3.5.2.2 command	10
3.5.2.3 key	10

3.5.2.4 key_size	10
3.5.2.5 key_size_buffer	10
3.5.2.6 parsing_index	10
3.5.2.7 parsing_stage	10
3.5.2.8 socket	11
3.5.2.9 value	11
3.5.2.10 value_size	11
3.5.2.11 value_size_buffer	11
3.6 HashNode Struct Reference	11
3.6.1 Detailed Description	11
3.6.2 Member Data Documentation	12
3.6.2.1 key	12
3.6.2.2 key_size	12
3.6.2.3 next	12
3.6.2.4 prev	12
3.6.2.5 prio	12
3.6.2.6 val	12
3.6.2.7 val_size	12
3.7 LookupResult Struct Reference	13
3.7.1 Detailed Description	13
3.7.2 Member Data Documentation	13
3.7.2.1 ptr	13
3.7.2.2 size	13
3.7.2.3 status	13
3.8 LRUNode Struct Reference	13
3.8.1 Detailed Description	14
3.8.2 Member Data Documentation	14
3.8.2.1 bucket_number	14
3.8.2.2 hash_node	14
3.8.2.3 next	14
3.8.2.4 prev	14
3.9 LRUQueue Struct Reference	14
3.9.1 Detailed Description	14
3.9.2 Member Data Documentation	15
3.9.2.1 least_recent	15
3.9.2.2 lock	15
3.9.2.3 most_recent	15
3.10 ServerArgs Struct Reference	15
3.10.1 Detailed Description	15
3.10.2 Member Data Documentation	15
3.10.2.1 cache	15
3.10.2.2 num_threads	16

3.10.2.3 server_epoll	16
3.10.2.4 server_socket	16
3.11 StatsReport Struct Reference	16
3.11.1 Detailed Description	16
3.11.2 Member Data Documentation	16
3.11.2.1 allocated_memory	16
3.11.2.2 del	17
3.11.2.3 evict	17
3.11.2.4 get	17
3.11.2.5 key	17
3.11.2.6 put	17
3.12 ThreadArgs Struct Reference	17
3.12.1 Detailed Description	18
3.12.2 Member Data Documentation	18
3.12.2.1 cache	18
3.12.2.2 server_epoll	18
3.12.2.3 server_socket	18
3.12.2.4 thread_id	18
3.12.2.5 thread_number	18
4 File Documentation	19
4.1 app/main.c File Reference	19
4.1.1 Macro Definition Documentation	20
4.1.1.1 GIGABYTE	20
4.1.1.2 KEY_COUNT	20
4.1.1.3 KEY_SIZE	20
4.1.1.4 MEGABYTE	20
4.1.1.5 MEMORY_LIMIT	20
4.1.1.6 VAL_COUNT	20
4.1.1.7 VAL_SIZE	20
4.1.2 Function Documentation	21
4.1.2.1 main()	21
4.1.2.2 set_memory_limit()	21
4.1.2.3 thread_func()	22
4.1.3 Variable Documentation	23
4.1.3.1 turnstile	23
4.2 main.c	23
4.3 app/main2.c File Reference	25
4.3.1 Macro Definition Documentation	26
4.3.1.1 GIGABYTE	26
4.3.1.2 KEY_COUNT	26
4.3.1.3 KEY_SIZE	26

4.3.1.4 MEGABYTE	26
4.3.1.5 MEMORY_LIMIT	27
4.3.1.6 VAL_COUNT	27
4.3.1.7 VAL_SIZE	27
4.3.2 Function Documentation	27
4.3.2.1 main()	27
4.3.2.2 set_memory_limit()	28
4.3.2.3 thread_func()	28
4.3.3 Variable Documentation	29
4.3.3.1 turnstile	29
4.4 main2.c	29
4.5 app/main3.c File Reference	31
4.5.1 Macro Definition Documentation	32
4.5.1.1 KEY_COUNT	32
4.5.1.2 KEY_SIZE	32
4.5.1.3 VAL_COUNT	32
4.5.1.4 VAL_SIZE	32
4.5.2 Function Documentation	33
4.5.2.1 main()	33
4.5.2.2 main2()	34
4.6 main3.c	34
4.7 cache/cache.c File Reference	35
4.7.1 Macro Definition Documentation	36
4.7.1.1 BUCKETS_FACTOR	36
4.7.1.2 ZONES_FACTOR	37
4.7.2 Function Documentation	37
4.7.2.1 cache_create()	37
4.7.2.2 cache_delete()	37
4.7.2.3 cache_destroy()	38
4.7.2.4 cache_free_up_memory()	39
4.7.2.5 cache_get()	40
4.7.2.6 cache_get_cstats()	41
4.7.2.7 cache_put()	42
4.7.2.8 cache_report()	43
4.8 cache.c	43
4.9 cache/cache.h File Reference	49
4.9.1 Macro Definition Documentation	50
4.9.1.1 DEBUG	50
4.9.1.2 PRINT	50
4.9.2 Typedef Documentation	50
4.9.2.1 Cache	50
4.9.2.2 HashFunction	51

4.9.3 Function Documentation	51
4.9.3.1 cache_create()	51
4.9.3.2 cache_delete()	51
4.9.3.3 cache_destroy()	52
4.9.3.4 cache_free_up_memory()	53
4.9.3.5 cache_get()	54
4.9.3.6 cache_get_cstats()	55
4.9.3.7 cache_put()	56
4.9.3.8 cache_report()	57
4.9.3.9 kr_hash()	57
4.10 cache.h	58
4.11 cache/cache_stats.c File Reference	59
4.11.1 Function Documentation	60
4.11.1.1 cache_stats_allocated_memory_add()	60
4.11.1.2 cache_stats_allocated_memory_free()	61
4.11.1.3 cache_stats_create()	61
4.11.1.4 cache_stats_del_counter_dec()	62
4.11.1.5 cache_stats_del_counter_inc()	62
4.11.1.6 cache_stats_destroy()	62
4.11.1.7 cache_stats_evict_counter_dec()	63
4.11.1.8 cache_stats_evict_counter_inc()	63
4.11.1.9 cache_stats_get_allocated_memory()	64
4.11.1.10 cache_stats_get_counter_dec()	64
4.11.1.11 cache_stats_get_counter_inc()	64
4.11.1.12 cache_stats_key_counter_dec()	65
4.11.1.13 cache_stats_key_counter_inc()	65
4.11.1.14 cache_stats_put_counter_dec()	66
4.11.1.15 cache_stats_put_counter_inc()	66
4.11.1.16 cache_stats_report()	66
4.11.1.17 cache_stats_show()	67
4.11.1.18 stats_report_stringify()	67
4.12 cache_stats.c	68
4.13 cache/cache_stats.h File Reference	70
4.13.1 Macro Definition Documentation	71
4.13.1.1 STATS_COUNT	71
4.13.1.2 STATS_MESSAGE_LENGTH	71
4.13.2 Typedef Documentation	71
4.13.2.1 CacheStats	71
4.13.2.2 StatsReport	72
4.13.3 Function Documentation	72
4.13.3.1 cache_stats_allocated_memory_add()	72
4.13.3.2 cache_stats_allocated_memory_free()	72

4.13.3.3 cache_stats_create()	73
4.13.3.4 cache_stats_del_counter_dec()	73
4.13.3.5 cache_stats_del_counter_inc()	73
4.13.3.6 cache_stats_destroy()	74
4.13.3.7 cache_stats_evict_counter_dec()	74
4.13.3.8 cache_stats_evict_counter_inc()	75
4.13.3.9 cache_stats_get_allocated_memory()	75
4.13.3.10 cache_stats_get_counter_dec()	76
4.13.3.11 cache_stats_get_counter_inc()	76
4.13.3.12 cache_stats_key_counter_dec()	76
4.13.3.13 cache_stats_key_counter_inc()	77
4.13.3.14 cache_stats_put_counter_dec()	77
4.13.3.15 cache_stats_put_counter_inc()	77
4.13.3.16 cache_stats_report()	79
4.13.3.17 cache_stats_show()	79
4.13.3.18 stats_report_stringify()	80
4.14 cache_stats.h	80
4.15 dynalloc/dynalloc.c File Reference	83
4.15.1 Macro Definition Documentation	83
4.15.1.1 DYNALLOC_FAIL_RATE	83
4.15.1.2 MAX_ATTEMPTS	83
4.15.1.3 SIZE_GOAL_FACTOR	83
4.15.2 Function Documentation	83
4.15.2.1 dynalloc()	83
4.16 dynalloc.c	84
4.17 dynalloc/dynalloc.h File Reference	85
4.17.1 Typedef Documentation	86
4.17.1.1 Cache	86
4.17.2 Function Documentation	86
4.17.2.1 dynalloc()	86
4.18 dynalloc.h	87
4.19 hashmap/hash.c File Reference	87
4.19.1 Function Documentation	87
4.19.1.1 dek_hash()	87
4.19.1.2 kr_hash()	88
4.20 hash.c	88
4.21 hashmap/hash.h File Reference	88
4.21.1 Function Documentation	89
4.21.1.1 dek_hash()	89
4.21.1.2 kr_hash()	89
4.22 hash.h	89
4.23 hashmap/hashnode.c File Reference	90

4.23.1 Function Documentation	91
4.23.1.1 hashnode_clean()	91
4.23.1.2 hashnode_create()	91
4.23.1.3 hashnode_destroy()	92
4.23.1.4 hashnode_get_key()	92
4.23.1.5 hashnode_get_key_size()	93
4.23.1.6 hashnode_get_next()	93
4.23.1.7 hashnode_get_prev()	94
4.23.1.8 hashnode_get_prio()	94
4.23.1.9 hashnode_get_val()	94
4.23.1.10 hashnode_get_val_size()	95
4.23.1.11 hashnode_keys_equal()	95
4.23.1.12 hashnode_lookup()	96
4.23.1.13 hashnode_lookup_node()	96
4.23.1.14 hashnode_set_key()	97
4.23.1.15 hashnode_set_key_size()	97
4.23.1.16 hashnode_set_next()	98
4.23.1.17 hashnode_set_prev()	98
4.23.1.18 hashnode_set_prio()	98
4.23.1.19 hashnode_set_val()	99
4.23.1.20 hashnode_set_val_size()	99
4.24 hashnode.c	100
4.25 hashmap/hashnode.h File Reference	102
4.25.1 Typedef Documentation	103
4.25.1.1 HashNode	103
4.25.1.2 LRUNode	104
4.25.2 Function Documentation	104
4.25.2.1 hashnode_clean()	104
4.25.2.2 hashnode_create()	104
4.25.2.3 hashnode_destroy()	105
4.25.2.4 hashnode_get_key()	106
4.25.2.5 hashnode_get_key_size()	106
4.25.2.6 hashnode_get_next()	106
4.25.2.7 hashnode_get_prev()	107
4.25.2.8 hashnode_get_prio()	107
4.25.2.9 hashnode_get_val()	108
4.25.2.10 hashnode_get_val_size()	108
4.25.2.11 hashnode_lookup()	108
4.25.2.12 hashnode_lookup_node()	109
4.25.2.13 hashnode_set_key()	110
4.25.2.14 hashnode_set_key_size()	110
4.25.2.15 hashnode_set_next()	110

4.25.2.16	hashnode_set_prev()	111
4.25.2.17	hashnode_set_prio()	111
4.25.2.18	hashnode_set_val()	112
4.25.2.19	hashnode_set_val_size()	112
4.26	hashnode.h	113
4.27	helpers/atom_counter.c File Reference	115
4.27.1	Function Documentation	116
4.27.1.1	atom_counter_add()	116
4.27.1.2	atom_counter_create()	116
4.27.1.3	atom_counter_dec()	116
4.27.1.4	atom_counter_destroy()	117
4.27.1.5	atom_counter_drop()	117
4.27.1.6	atom_counter_get()	117
4.27.1.7	atom_counter_inc()	118
4.28	atom_counter.c	118
4.29	helpers/atom_counter.h File Reference	119
4.29.1	Macro Definition Documentation	120
4.29.1.1	COUNTER_FORMAT	120
4.29.2	Typedef Documentation	120
4.29.2.1	AtomCounter	120
4.29.2.2	Counter	120
4.29.3	Function Documentation	120
4.29.3.1	atom_counter_add()	120
4.29.3.2	atom_counter_create()	121
4.29.3.3	atom_counter_dec()	121
4.29.3.4	atom_counter_destroy()	121
4.29.3.5	atom_counter_drop()	122
4.29.3.6	atom_counter_get()	122
4.29.3.7	atom_counter_inc()	122
4.30	atom_counter.h	123
4.31	helpers/quit.c File Reference	123
4.31.1	Function Documentation	123
4.31.1.1	quit()	123
4.32	quit.c	124
4.33	helpers/quit.h File Reference	124
4.33.1	Function Documentation	124
4.33.1.1	quit()	124
4.34	quit.h	124
4.35	helpers/results.c File Reference	125
4.35.1	Function Documentation	125
4.35.1.1	create_error_lookup_result()	125
4.35.1.2	create_miss_lookup_result()	125

4.35.1.3	create_ok_lookup_result()	126
4.35.1.4	lookup_result_get_size()	126
4.35.1.5	lookup_result_get_value()	126
4.35.1.6	lookup_result_is_error()	126
4.35.1.7	lookup_result_is_miss()	126
4.35.1.8	lookup_result_is_ok()	127
4.36	results.c	127
4.37	helpers/results.h File Reference	127
4.37.1	Typedef Documentation	128
4.37.1.1	LookupResult	128
4.37.2	Enumeration Type Documentation	128
4.37.2.1	Status	128
4.37.3	Function Documentation	129
4.37.3.1	create_error_lookup_result()	129
4.37.3.2	create_miss_lookup_result()	129
4.37.3.3	create_ok_lookup_result()	129
4.37.3.4	lookup_result_get_size()	129
4.37.3.5	lookup_result_get_value()	129
4.37.3.6	lookup_result_is_error()	130
4.37.3.7	lookup_result_is_miss()	130
4.37.3.8	lookup_result_is_ok()	130
4.38	results.h	130
4.39	lru/lru.c File Reference	131
4.39.1	Function Documentation	132
4.39.1.1	lru_queue_create()	132
4.39.1.2	lru_queue_delete_node()	132
4.39.1.3	lru_queue_destroy()	133
4.39.1.4	lru_queue_get_least_recent()	133
4.39.1.5	lru_queue_lock()	134
4.39.1.6	Funciones auxiliares	134
4.39.1.7	lru_queue_node_clean()	134
4.39.1.8	lru_queue_set_most_recent()	135
4.39.1.9	lru_queue_unlock()	135
4.40	lru.c	136
4.41	lru/lru.h File Reference	138
4.41.1	Typedef Documentation	138
4.41.1.1	HashNode	138
4.41.1.2	LRUNode	138
4.41.1.3	LRUQueue	138
4.41.2	Function Documentation	138
4.41.2.1	lru_queue_create()	138
4.41.2.2	lru_queue_delete_node()	139

4.41.2.3	lru_queue_destroy()	139
4.41.2.4	lru_queue_get_least_recent()	140
4.41.2.5	lru_queue_lock()	140
4.41.2.6	Funciones auxiliares	141
4.41.2.7	lru_queue_node_clean()	141
4.41.2.8	lru_queue_set_most_recent()	141
4.41.2.9	lru_queue_unlock()	142
4.42	lru.h	142
4.43	lru/lrunode.c File Reference	143
4.43.1	Macro Definition Documentation	144
4.43.1.1	PRINT	144
4.43.2	Function Documentation	144
4.43.2.1	lru_node_is_clean()	144
4.43.2.2	lrunode_create()	145
4.43.2.3	lrunode_destroy()	145
4.43.2.4	lrunode_get_bucket_number()	145
4.43.2.5	lrunode_get_hash_node()	146
4.43.2.6	lrunode_get_next()	146
4.43.2.7	lrunode_get_prev()	146
4.43.2.8	lrunode_set_bucket_number()	147
4.43.2.9	lrunode_set_hash_node()	147
4.43.2.10	lrunode_set_next()	147
4.43.2.11	lrunode_set_prev()	148
4.44	lrunode.c	148
4.45	lru/lrunode.h File Reference	149
4.45.1	Typedef Documentation	150
4.45.1.1	Cache	150
4.45.1.2	HashNode	150
4.45.1.3	LRUNode	150
4.45.2	Function Documentation	150
4.45.2.1	lru_node_is_clean()	150
4.45.2.2	lrunode_create()	150
4.45.2.3	lrunode_destroy()	151
4.45.2.4	lrunode_get_bucket_number()	151
4.45.2.5	lrunode_get_hash_node()	151
4.45.2.6	lrunode_get_next()	152
4.45.2.7	lrunode_get_prev()	152
4.45.2.8	lrunode_set_bucket_number()	152
4.45.2.9	lrunode_set_hash_node()	153
4.45.2.10	lrunode_set_next()	153
4.45.2.11	lrunode_set_prev()	153
4.46	lrunode.h	154

4.47 server/cache_server.c File Reference	155
4.47.1 Macro Definition Documentation	155
4.47.1.1 BLUE	155
4.47.1.2 GREEN	155
4.47.1.3 ORANGE	155
4.47.1.4 RED	155
4.47.1.5 RESET	156
4.47.1.6 SOFT_RED	156
4.47.2 Function Documentation	156
4.47.2.1 main()	156
4.47.2.2 start_server()	156
4.47.2.3 working_thread()	157
4.48 cache_server.c	158
4.49 server/cache_server_models.h File Reference	161
4.49.1 Macro Definition Documentation	162
4.49.1.1 LENGTH_PREFIX_SIZE	162
4.49.2 Enumeration Type Documentation	162
4.49.2.1 Command	162
4.49.2.2 ParsingStage	162
4.49.2.3 Response	162
4.50 cache_server_models.h	163
4.51 server/cache_server_utils.c File Reference	164
4.51.1 Macro Definition Documentation	165
4.51.1.1 TRASH_BUFFER_SIZE	165
4.51.2 Function Documentation	165
4.51.2.1 clean_socket()	165
4.51.2.2 construct_new_client_epoll()	165
4.51.2.3 create_new_client_data()	166
4.51.2.4 delete_client_data()	166
4.51.2.5 drop_client()	167
4.51.2.6 handle_request()	167
4.51.2.7 parse_request()	169
4.51.2.8 reconstruct_client_epoll()	171
4.51.2.9 rcv_client()	171
4.51.2.10 reset_client_data()	172
4.51.2.11 send_client()	172
4.52 cache_server_utils.c	173
4.53 server/cache_server_utils.h File Reference	178
4.53.1 Function Documentation	179
4.53.1.1 clean_socket()	179
4.53.1.2 construct_new_client_epoll()	179
4.53.1.3 create_new_client_data()	180

4.53.1.4 delete_client_data()	180
4.53.1.5 drop_client()	181
4.53.1.6 handle_request()	181
4.53.1.7 parse_request()	183
4.53.1.8 reconstruct_client_epoll()	184
4.53.1.9 recv_client()	185
4.53.1.10 reset_client_data()	185
4.53.1.11 send_client()	186
4.54 cache_server_utils.h	186
4.55 server/server_starter.c File Reference	188
4.55.1 Function Documentation	189
4.55.1.1 main()	189
4.56 server_starter.c	189
4.57 server/server_starter_utils.c File Reference	190
4.57.1 Macro Definition Documentation	190
4.57.1.1 BACKLOG_SIZE	190
4.57.1.2 DEFAULT_PORT	190
4.57.2 Function Documentation	190
4.57.2.1 create_server_socket()	190
4.57.2.2 exec_server()	191
4.57.2.3 parse_arguments()	191
4.57.2.4 set_memory_limit()	192
4.58 server_starter_utils.c	193
4.59 server/server_starter_utils.h File Reference	195
4.59.1 Macro Definition Documentation	195
4.59.1.1 DEFAULT_PORT	195
4.59.2 Function Documentation	195
4.59.2.1 create_server_socket()	195
4.59.2.2 exec_server()	196
4.59.2.3 parse_arguments()	196
4.59.2.4 set_memory_limit()	197
4.60 server_starter_utils.h	198

Chapter 1

Class Index

1.1 Class List

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

Args	5
AtomCounter	6
Cache	6
CacheStats	8
ClientData	9
HashNode	11
LookupResult	13
LRUNode	13
LRUQueue	14
ServerArgs	15
StatsReport	16
ThreadArgs	17

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

app/main.c	19
app/main2.c	25
app/main3.c	31
cache/cache.c	35
cache/cache.h	49
cache/cache_stats.c	59
cache/cache_stats.h	70
dynalloc/dynalloc.c	83
dynalloc/dynalloc.h	85
hashmap/hash.c	87
hashmap/hash.h	88
hashmap/hashnode.c	90
hashmap/hashnode.h	102
helpers/atom_counter.c	115
helpers/atom_counter.h	119
helpers/quit.c	123
helpers/quit.h	124
helpers/results.c	125
helpers/results.h	127
lru/lru.c	131
lru/lru.h	138
lru/lrunode.c	143
lru/lrunode.h	149
server/cache_server.c	155
server/cache_server_models.h	161
server/cache_server_utils.c	164
server/cache_server_utils.h	178
server/server_starter.c	188
server/server_starter_utils.c	190
server/server_starter_utils.h	195

Chapter 3

Class Documentation

3.1 Args Struct Reference

```
#include <server_starter_utils.h>
```

Public Attributes

- int [port](#)
- unsigned long [memory_limit](#)
- int [num_threads](#)

3.1.1 Detailed Description

Definition at line [21](#) of file [server_starter_utils.h](#).

3.1.2 Member Data Documentation

3.1.2.1 [memory_limit](#)

```
unsigned long Args::memory_limit
```

Definition at line [24](#) of file [server_starter_utils.h](#).

3.1.2.2 [num_threads](#)

```
int Args::num_threads
```

Definition at line [25](#) of file [server_starter_utils.h](#).

3.1.2.3 port

```
int Args::port
```

Definition at line 23 of file [server_starter_utils.h](#).

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

- [server/server_starter_utils.h](#)

3.2 AtomCounter Struct Reference

Public Attributes

- [Counter](#) counter
- [pthread_rwlock_t](#) lock

3.2.1 Detailed Description

Definition at line 5 of file [atom_counter.c](#).

3.2.2 Member Data Documentation

3.2.2.1 counter

```
Counter AtomCounter::counter
```

Definition at line 7 of file [atom_counter.c](#).

3.2.2.2 lock

```
pthread\_rwlock\_t AtomCounter::lock
```

Definition at line 9 of file [atom_counter.c](#).

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

- [helpers/atom_counter.c](#)

3.3 Cache Struct Reference

Public Attributes

- [HashFunction](#) hash_function
- [HashNode](#) * buckets
- int num_buckets
- [pthread_rwlock_t](#) ** zone_locks
- int num_zones
- [LRUQueue](#) queue
- [CacheStats](#) stats

3.3.1 Detailed Description

Definition at line 14 of file [cache.c](#).

3.3.2 Member Data Documentation

3.3.2.1 buckets

[HashNode*](#) [Cache::buckets](#)

Definition at line 18 of file [cache.c](#).

3.3.2.2 hash_function

[HashFunction](#) [Cache::hash_function](#)

Definition at line 17 of file [cache.c](#).

3.3.2.3 num_buckets

[int](#) [Cache::num_buckets](#)

Definition at line 19 of file [cache.c](#).

3.3.2.4 num_zones

[int](#) [Cache::num_zones](#)

Definition at line 21 of file [cache.c](#).

3.3.2.5 queue

[LRUQueue](#) [Cache::queue](#)

Definition at line 24 of file [cache.c](#).

3.3.2.6 stats

[CacheStats](#) [Cache::stats](#)

Definition at line 27 of file [cache.c](#).

3.3.2.7 zone_locks

`pthread_rwlock_t** Cache::zone_locks`

Definition at line 20 of file [cache.c](#).

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

- [cache/cache.c](#)

3.4 CacheStats Struct Reference

Public Attributes

- [AtomCounter](#) put_counter
- [AtomCounter](#) get_counter
- [AtomCounter](#) del_counter
- [AtomCounter](#) evict_counter
- [AtomCounter](#) key_counter
- [AtomCounter](#) allocated_memory

3.4.1 Detailed Description

Definition at line 6 of file [cache_stats.c](#).

3.4.2 Member Data Documentation

3.4.2.1 allocated_memory

[AtomCounter](#) `CacheStats::allocated_memory`

Definition at line 15 of file [cache_stats.c](#).

3.4.2.2 del_counter

[AtomCounter](#) `CacheStats::del_counter`

Definition at line 10 of file [cache_stats.c](#).

3.4.2.3 evict_counter

[AtomCounter](#) `CacheStats::evict_counter`

Definition at line 12 of file [cache_stats.c](#).

3.4.2.4 get_counter

`AtomCounter` `CacheStats::get_counter`

Definition at line 9 of file `cache_stats.c`.

3.4.2.5 key_counter

`AtomCounter` `CacheStats::key_counter`

Definition at line 13 of file `cache_stats.c`.

3.4.2.6 put_counter

`AtomCounter` `CacheStats::put_counter`

Definition at line 8 of file `cache_stats.c`.

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

- `cache/cache_stats.c`

3.5 ClientData Struct Reference

```
#include <cache_server_models.h>
```

Public Attributes

- `int` `socket`
- `char` `command`
- `char` `key_size_buffer` [`LENGTH_PREFIX_SIZE`]
- `int` `key_size`
- `char *` `key`
- `char` `value_size_buffer` [`LENGTH_PREFIX_SIZE`]
- `int` `value_size`
- `char *` `value`
- `int` `parsing_index`
- `ParsingStage` `parsing_stage`
- `int` `cleaning`

3.5.1 Detailed Description

Definition at line 56 of file `cache_server_models.h`.

3.5.2 Member Data Documentation

3.5.2.1 cleaning

```
int ClientData::cleaning
```

Definition at line 73 of file [cache_server_models.h](#).

3.5.2.2 command

```
char ClientData::command
```

Definition at line 60 of file [cache_server_models.h](#).

3.5.2.3 key

```
char* ClientData::key
```

Definition at line 64 of file [cache_server_models.h](#).

3.5.2.4 key_size

```
int ClientData::key_size
```

Definition at line 63 of file [cache_server_models.h](#).

3.5.2.5 key_size_buffer

```
char ClientData::key_size_buffer[LENGTH_PREFIX_SIZE]
```

Definition at line 62 of file [cache_server_models.h](#).

3.5.2.6 parsing_index

```
int ClientData::parsing_index
```

Definition at line 70 of file [cache_server_models.h](#).

3.5.2.7 parsing_stage

```
ParsingStage ClientData::parsing_stage
```

Definition at line 71 of file [cache_server_models.h](#).

3.5.2.8 socket

```
int ClientData::socket
```

Definition at line 58 of file [cache_server_models.h](#).

3.5.2.9 value

```
char* ClientData::value
```

Definition at line 68 of file [cache_server_models.h](#).

3.5.2.10 value_size

```
int ClientData::value_size
```

Definition at line 67 of file [cache_server_models.h](#).

3.5.2.11 value_size_buffer

```
char ClientData::value_size_buffer[LENGTH_PREFIX_SIZE]
```

Definition at line 66 of file [cache_server_models.h](#).

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

- [server/cache_server_models.h](#)

3.6 HashNode Struct Reference

Public Attributes

- void * [key](#)
- void * [val](#)
- size_t [key_size](#)
- size_t [val_size](#)
- struct [HashNode](#) * [prev](#)
- struct [HashNode](#) * [next](#)
- struct [LRUNode](#) * [prio](#)

3.6.1 Detailed Description

Definition at line 8 of file [hashnode.c](#).

3.6.2 Member Data Documentation

3.6.2.1 key

```
void* HashNode::key
```

Definition at line 9 of file [hashnode.c](#).

3.6.2.2 key_size

```
size_t HashNode::key_size
```

Definition at line 11 of file [hashnode.c](#).

3.6.2.3 next

```
struct HashNode* HashNode::next
```

Definition at line 14 of file [hashnode.c](#).

3.6.2.4 prev

```
struct HashNode* HashNode::prev
```

Definition at line 13 of file [hashnode.c](#).

3.6.2.5 prio

```
struct LRUNode* HashNode::prio
```

Definition at line 15 of file [hashnode.c](#).

3.6.2.6 val

```
void* HashNode::val
```

Definition at line 10 of file [hashnode.c](#).

3.6.2.7 val_size

```
size_t HashNode::val_size
```

Definition at line 11 of file [hashnode.c](#).

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

- [hashmap/hashnode.c](#)

3.7 LookupResult Struct Reference

```
#include <results.h>
```

Public Attributes

- void * [ptr](#)
- size_t [size](#)
- [Status](#) [status](#)

3.7.1 Detailed Description

Definition at line 10 of file [results.h](#).

3.7.2 Member Data Documentation

3.7.2.1 ptr

```
void* LookupResult::ptr
```

Definition at line 12 of file [results.h](#).

3.7.2.2 size

```
size_t LookupResult::size
```

Definition at line 13 of file [results.h](#).

3.7.2.3 status

```
Status LookupResult::status
```

Definition at line 15 of file [results.h](#).

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

- [helpers/results.h](#)

3.8 LRUNode Struct Reference

Public Attributes

- struct [LRUNode](#) * [prev](#)
- struct [LRUNode](#) * [next](#)
- [HashNode](#) [hash_node](#)
- unsigned int [bucket_number](#)

3.8.1 Detailed Description

Definition at line 9 of file [lrunode.c](#).

3.8.2 Member Data Documentation

3.8.2.1 bucket_number

```
unsigned int LRUNode::bucket_number
```

Definition at line 14 of file [lrunode.c](#).

3.8.2.2 hash_node

```
HashNode LRUNode::hash_node
```

Definition at line 13 of file [lrunode.c](#).

3.8.2.3 next

```
struct LRUNode* LRUNode::next
```

Definition at line 11 of file [lrunode.c](#).

3.8.2.4 prev

```
struct LRUNode* LRUNode::prev
```

Definition at line 10 of file [lrunode.c](#).

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

- [lru/lrunode.c](#)

3.9 LRUQueue Struct Reference

Public Attributes

- [LRUNode](#) most_recent
- [LRUNode](#) least_recent
- pthread_mutex_t * lock

3.9.1 Detailed Description

Definition at line 9 of file [lru.c](#).

3.9.2 Member Data Documentation

3.9.2.1 least_recent

[LRUNode](#) [LRUQueue::least_recent](#)

Definition at line 12 of file [lru.c](#).

3.9.2.2 lock

[pthread_mutex_t*](#) [LRUQueue::lock](#)

Definition at line 14 of file [lru.c](#).

3.9.2.3 most_recent

[LRUNode](#) [LRUQueue::most_recent](#)

Definition at line 11 of file [lru.c](#).

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

- [lru/lru.c](#)

3.10 ServerArgs Struct Reference

```
#include <cache_server_models.h>
```

Public Attributes

- int [server_epoll](#)
- int [server_socket](#)
- int [num_threads](#)
- [Cache](#) [cache](#)

3.10.1 Detailed Description

Definition at line 47 of file [cache_server_models.h](#).

3.10.2 Member Data Documentation

3.10.2.1 cache

[Cache](#) [ServerArgs::cache](#)

Definition at line 52 of file [cache_server_models.h](#).

3.10.2.2 num_threads

```
int ServerArgs::num_threads
```

Definition at line 51 of file [cache_server_models.h](#).

3.10.2.3 server_epoll

```
int ServerArgs::server_epoll
```

Definition at line 49 of file [cache_server_models.h](#).

3.10.2.4 server_socket

```
int ServerArgs::server_socket
```

Definition at line 50 of file [cache_server_models.h](#).

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

- [server/cache_server_models.h](#)

3.11 StatsReport Struct Reference

```
#include <cache_stats.h>
```

Public Attributes

- [Counter put](#)
- [Counter get](#)
- [Counter del](#)
- [Counter key](#)
- [Counter evict](#)
- [Counter allocated_memory](#)

3.11.1 Detailed Description

Definition at line 11 of file [cache_stats.h](#).

3.11.2 Member Data Documentation

3.11.2.1 allocated_memory

```
Counter StatsReport::allocated_memory
```

Definition at line 17 of file [cache_stats.h](#).

3.11.2.2 del

`Counter StatsReport::del`

Definition at line 14 of file [cache_stats.h](#).

3.11.2.3 evict

`Counter StatsReport::evict`

Definition at line 16 of file [cache_stats.h](#).

3.11.2.4 get

`Counter StatsReport::get`

Definition at line 13 of file [cache_stats.h](#).

3.11.2.5 key

`Counter StatsReport::key`

Definition at line 15 of file [cache_stats.h](#).

3.11.2.6 put

`Counter StatsReport::put`

Definition at line 12 of file [cache_stats.h](#).

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

- [cache/cache_stats.h](#)

3.12 ThreadArgs Struct Reference

```
#include <cache_server_models.h>
```

Public Attributes

- `int` [thread_id](#)
- `Cache` [cache](#)
- `int` [server_epoll](#)
- `int` [server_socket](#)
- `int` [thread_number](#)

3.12.1 Detailed Description

Definition at line 21 of file [main.c](#).

3.12.2 Member Data Documentation

3.12.2.1 cache

```
Cache ThreadArgs::cache
```

Definition at line 23 of file [main.c](#).

3.12.2.2 server_epoll

```
int ThreadArgs::server_epoll
```

Definition at line 40 of file [cache_server_models.h](#).

3.12.2.3 server_socket

```
int ThreadArgs::server_socket
```

Definition at line 41 of file [cache_server_models.h](#).

3.12.2.4 thread_id

```
int ThreadArgs::thread_id
```

Definition at line 22 of file [main.c](#).

3.12.2.5 thread_number

```
int ThreadArgs::thread_number
```

Definition at line 42 of file [cache_server_models.h](#).

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

- [app/main.c](#)
- [app/main2.c](#)
- [server/cache_server_models.h](#)

Chapter 4

File Documentation

4.1 app/main.c File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <time.h>
#include <sys/resource.h>
#include <semaphore.h>
#include "cache/cache.h"
```

Classes

- struct [ThreadArgs](#)

Macros

- #define [KEY_SIZE](#) 12
- #define [KEY_COUNT](#) 1000
- #define [VAL_SIZE](#) sizeof(int)
- #define [VAL_COUNT](#) 150
- #define [MEGABYTE](#) 1000000
- #define [GIGABYTE](#) 1000 * [MEGABYTE](#)
- #define [MEMORY_LIMIT](#) 116 * [MEGABYTE](#)

Functions

- void * [thread_func](#) (void *arg)
- void [set_memory_limit](#) (size_t limit_bytes)
- int [main](#) ()

Variables

- sem_t [turnstile](#)

4.1.1 Macro Definition Documentation

4.1.1.1 GIGABYTE

```
#define GIGABYTE 1000 * MEGABYTE
```

Definition at line 17 of file [main.c](#).

4.1.1.2 KEY_COUNT

```
#define KEY_COUNT 1000
```

Definition at line 12 of file [main.c](#).

4.1.1.3 KEY_SIZE

```
#define KEY_SIZE 12
```

Definition at line 11 of file [main.c](#).

4.1.1.4 MEGABYTE

```
#define MEGABYTE 1000000
```

Definition at line 16 of file [main.c](#).

4.1.1.5 MEMORY_LIMIT

```
#define MEMORY_LIMIT 116 * MEGABYTE
```

Definition at line 18 of file [main.c](#).

4.1.1.6 VAL_COUNT

```
#define VAL_COUNT 150
```

Definition at line 14 of file [main.c](#).

4.1.1.7 VAL_SIZE

```
#define VAL_SIZE sizeof(int)
```

Definition at line 13 of file [main.c](#).

4.1.2 Function Documentation

4.1.2.1 main()

```
int main ( )
```

Definition at line 135 of file [main.c](#).

```
00135     {
00136
00137         setbuf(stdout, NULL);
00138
00139         set_memory_limit(MEMORY_LIMIT);
00140
00141         sem_init(&turnstile, 0, 1);
00142         sem_wait(&turnstile);
00143
00144         // Create the cache with a larger size to handle more entries
00145         Cache cache = cache_create((HashFunction)kr_hash);
00146         if (!cache) {
00147             PRINT("Failed to create cache");
00148             return 1;
00149         }
00150
00151         const int NUM_THREADS = 9;
00152         pthread_t threads[NUM_THREADS];
00153         ThreadArgs thread_args[NUM_THREADS];
00154
00155         // Create threads
00156         for (int i = 0; i < NUM_THREADS; i++) {
00157             thread_args[i].thread_id = i;
00158             thread_args[i].cache = cache;
00159
00160             int result = pthread_create(&threads[i], NULL, thread_func, &thread_args[i]);
00161             if (result != 0) {
00162                 PRINT("Failed to create thread %d", i);
00163                 return 1;
00164             }
00165         }
00166
00167         sem_post(&turnstile);
00168
00169         // Wait for all threads to finish
00170         for (int i = 0; i < NUM_THREADS; i++) {
00171             pthread_join(threads[i], NULL);
00172         }
00173
00174         // Show cache statistics
00175         cache_stats(cache);
00176
00177         // And destroy it.
00178         cache_destroy(cache);
00179
00180         return 0;
00181 }
```

4.1.2.2 set_memory_limit()

```
void set_memory_limit (
    size_t limit_bytes )
```

Definition at line 123 of file [main.c](#).

```
00123     {
00124         struct rlimit limit;
00125         limit.rlim_cur = limit_bytes; // Soft limit
00126         limit.rlim_max = limit_bytes; // Hard limit
00127
00128         if (setrlimit(RLIMIT_AS, &limit) != 0) {
00129             perror("setrlimit failed");
00130             exit(EXIT_FAILURE);
00131         }
00132 }
```

4.1.2.3 thread_func()

```
void * thread_func (
    void * arg )
```

Definition at line 30 of file [main.c](#).

```
00030         {
00031
00032         // Initial turnstile so it does not run out of memory before creating threads.
00033         sem_wait(&turnstile);
00034         sem_post(&turnstile);
00035
00036         ThreadArgs* args = (ThreadArgs*)arg;
00037         int thread_id = args->thread_id;
00038         Cache cache = args->cache;
00039
00040         char* keys[KEY_COUNT];
00041         char* keysGet[KEY_COUNT];
00042         int* vals[VAL_COUNT];
00043
00044         // Generate unique keys and values for this thread
00045         for (int i = 0; i < KEY_COUNT; i++) {
00046
00047             keys[i] = dynalloc(KEY_SIZE, cache);
00048             if (!keys[i]) {
00049                 PRINT("Thread %d: failed to allocate memory for key %d", thread_id, i);
00050                 return NULL;
00051             }
00052
00053             keysGet[i] = dynalloc(KEY_SIZE, cache);
00054             if (!keysGet[i]) {
00055                 PRINT("Thread %d: failed to allocate memory for key %d", thread_id, i);
00056                 return NULL;
00057             }
00058
00059             vals[i] = dynalloc(VAL_SIZE, cache);
00060             if (!vals[i]) {
00061                 PRINT("Thread %d: failed to allocate memory for value %d", thread_id, i);
00062                 free(keys[i]); // Free the key if value allocation fails
00063                 return NULL;
00064             }
00065
00066             // Generate a unique key for this thread
00067             snprintf(keys[i], KEY_SIZE, "key_%d_%d", thread_id, i);
00068             snprintf(keysGet[i], KEY_SIZE, "key_%d_%d", thread_id, i);
00069             *vals[i] = thread_id * KEY_COUNT + i; // Unique value for each key
00070
00071
00072             // ! Hago strlen sobre keysGet[i] porque puede que para cuando haga ese strlen ya me hayan
00073             tenido que liberar el keys[i].
00074             int result = cache_put(keys[i], strlen(keysGet[i]), vals[i], VAL_SIZE, cache);
00075             PRINT("Inserte el par clave valor numero %i.", i);
00076             if (result != 0) {
00077                 PRINT("Thread %d: failed to insert key-value pair: %s / %i", thread_id, keys[i],
00078                     *vals[i]);
00079             }
00080
00081             if (i == KEY_COUNT / 2)
00082                 cache_stats(cache);
00083
00084             cache_stats(cache);
00085
00086
00087         // // Retrieve keys from the cache
00088         // for (int i = 0; i < KEY_COUNT; i++) {
00089         //     LookupResult lr = cache_get(keysGet[i], strlen(keysGet[i]), cache);
00090
00091         //     /*
00092         //         if (lookup_result_is_ok(lr)) {
00093         //             int retrieved_value = *((int*)lookup_result_get_value(lr));
00094
00095         //             // ! Esto puede dar segfault si ya se libero vals[i].
00096         //             if (retrieved_value != *vals[i]) {
00097         //                 PRINT("Thread %d: cGET mismatch for key %s: expected %i, got %i", thread_id,
00098         //                     keys[i], *vals[i], retrieved_value);
00099         //             }
00100
00101         //             // PRINT("Thread %d: got the pair %s / %i", thread_id, keys[i], retrieved_value);
00102
00103         //         } else {
```

```

00105     //          PRINT("Thread %d: Failed to GET the key %s", thread_id, keys[i]);
00106     //      } */
00107     // }
00108
00109
00110     // Destroy all entries.
00111     /*
00112     for (int i = 0; i < KEY_COUNT; i++) {
00113         int result = cache_delete(keys[i], strlen(keys[i]), cache);
00114         if (result < 0) {
00115             PRINT("Thread %d: failed to delete key: %s", thread_id, keys[i]);
00116         }
00117     }
00118     */
00119
00120     return NULL;
00121 }

```

4.1.3 Variable Documentation

4.1.3.1 turnstile

sem_t turnstile

Definition at line 27 of file [main.c](#).

4.2 main.c

[Go to the documentation of this file.](#)

```

00001 #include <stdio.h>
00002 #include <stdlib.h>
00003 #include <string.h>
00004 #include <pthread.h>
00005 #include <time.h>
00006 #include <sys/resource.h>
00007 #include <semaphore.h>
00008
00009 #include "cache/cache.h"
00010
00011 #define KEY_SIZE 12
00012 #define KEY_COUNT 1000
00013 #define VAL_SIZE sizeof(int)
00014 #define VAL_COUNT 150
00015
00016 #define MEGABYTE 1000000
00017 #define GIGABYTE 1000 * MEGABYTE
00018 #define MEMORY_LIMIT 116 * MEGABYTE
00019
00020 // Struct to pass multiple arguments to the thread function
00021 typedef struct {
00022     int thread_id;
00023     Cache cache; // Cache is already a pointer (struct Cache*)
00024 } ThreadArgs;
00025
00026
00027 sem_t turnstile;
00028
00029 // Thread function to perform cache operations
00030 void* thread_func(void* arg) {
00031
00032     // Initial turnstile so it does not run out of memory before creating threads.
00033     sem_wait(&turnstile);
00034     sem_post(&turnstile);
00035
00036     ThreadArgs* args = (ThreadArgs*)arg;
00037     int thread_id = args->thread_id;
00038     Cache cache = args->cache;
00039
00040     char* keys[KEY_COUNT];
00041     char* keysGet[KEY_COUNT];
00042     int* vals[VAL_COUNT];
00043
00044     // Generate unique keys and values for this thread

```

```

00045     for (int i = 0; i < KEY_COUNT; i++) {
00046
00047         keys[i] = dynalloc(KEY_SIZE, cache);
00048         if (!keys[i]) {
00049             PRINT("Thread %d: failed to allocate memory for key %d", thread_id, i);
00050             return NULL;
00051         }
00052
00053         keysGet[i] = dynalloc(KEY_SIZE, cache);
00054         if (!keysGet[i]) {
00055             PRINT("Thread %d: failed to allocate memory for key %d", thread_id, i);
00056             return NULL;
00057         }
00058
00059         vals[i] = dynalloc(VAL_SIZE, cache);
00060         if (!vals[i]) {
00061             PRINT("Thread %d: failed to allocate memory for value %d", thread_id, i);
00062             free(keys[i]); // Free the key if value allocation fails
00063             return NULL;
00064         }
00065
00066         // Generate a unique key for this thread
00067         snprintf(keys[i], KEY_SIZE, "key_%d_%d", thread_id, i);
00068         snprintf(keysGet[i], KEY_SIZE, "key_%d_%d", thread_id, i);
00069         *vals[i] = thread_id * KEY_COUNT + i; // Unique value for each key
00070
00071
00072         // ! Hago strlen sobre keysGet[i] porque puede que para cuando haga ese strlen ya me hayan
00073         tenido que liberar el keys[i].
00074         int result = cache_put(keys[i], strlen(keysGet[i]), vals[i], VAL_SIZE, cache);
00075         PRINT("Inserte el par clave valor numero %i.", i);
00076         if (result != 0) {
00077             PRINT("Thread %d: failed to insert key-value pair: %s / %i", thread_id, keys[i],
00078                 *vals[i]);
00079         }
00080
00081         if (i == KEY_COUNT / 2)
00082             cache_stats(cache);
00083     }
00084
00085     cache_stats(cache);
00086
00087
00088
00089     // // Retrieve keys from the cache
00090     // for (int i = 0; i < KEY_COUNT; i++) {
00091     //     LookupResult lr = cache_get(keysGet[i], strlen(keysGet[i]), cache);
00092
00093     //     /*
00094     //      * if (lookup_result_is_ok(lr)) {
00095     //          int retrieved_value = *((int*)lookup_result_get_value(lr));
00096
00097     //          // ! Esto puede dar segfault si ya se libero vals[i].
00098     //          if (retrieved_value != *vals[i]) {
00099     //              PRINT("Thread %d: cGET mismatch for key %s: expected %i, got %i", thread_id,
00100     //                  keys[i], *vals[i], retrieved_value);
00101     //          }
00102     //          // PRINT("Thread %d: got the pair %s / %i", thread_id, keys[i], retrieved_value);
00103
00104     //      } else {
00105     //          PRINT("Thread %d: Failed to GET the key %s", thread_id, keys[i]);
00106     //      } */
00107     // }
00108
00109
00110     // Destroy all entries.
00111     /*
00112     for (int i = 0; i < KEY_COUNT; i++) {
00113         int result = cache_delete(keys[i], strlen(keys[i]), cache);
00114         if (result < 0) {
00115             PRINT("Thread %d: failed to delete key: %s", thread_id, keys[i]);
00116         }
00117     }
00118     */
00119
00120     return NULL;
00121 }
00122
00123 void set_memory_limit(size_t limit_bytes) {
00124     struct rlimit limit;
00125     limit.rlim_cur = limit_bytes; // Soft limit
00126     limit.rlim_max = limit_bytes; // Hard limit
00127
00128     if (setrlimit(RLIMIT_AS, &limit) != 0) {

```

```

00129         perror("setrlimit failed");
00130         exit(EXIT_FAILURE);
00131     }
00132 }
00133
00134
00135 int main() {
00136     setbuf(stdout, NULL);
00137
00138     set_memory_limit(MEMORY_LIMIT);
00139
00140     sem_init(&turnstile, 0, 1);
00141     sem_wait(&turnstile);
00142
00143     // Create the cache with a larger size to handle more entries
00144     Cache cache = cache_create((HashFunction)kr_hash);
00145     if (!cache) {
00146         PRINT("Failed to create cache");
00147         return 1;
00148     }
00149
00150     const int NUM_THREADS = 9;
00151     pthread_t threads[NUM_THREADS];
00152     ThreadArgs thread_args[NUM_THREADS];
00153
00154     // Create threads
00155     for (int i = 0; i < NUM_THREADS; i++) {
00156         thread_args[i].thread_id = i;
00157         thread_args[i].cache = cache;
00158
00159         int result = pthread_create(&threads[i], NULL, thread_func, &thread_args[i]);
00160         if (result != 0) {
00161             PRINT("Failed to create thread %d", i);
00162             return 1;
00163         }
00164     }
00165
00166     sem_post(&turnstile);
00167
00168     // Wait for all threads to finish
00169     for (int i = 0; i < NUM_THREADS; i++) {
00170         pthread_join(threads[i], NULL);
00171     }
00172
00173     // Show cache statistics
00174     cache_stats(cache);
00175
00176     // And destroy it.
00177     cache_destroy(cache);
00178
00179     return 0;
00180 }
00181 }

```

4.3 app/main2.c File Reference

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <time.h>
#include <sys/resource.h>
#include <semaphore.h>
#include "cache/cache.h"

```

Classes

- struct [ThreadArgs](#)

Macros

- `#define KEY_SIZE 12`
- `#define KEY_COUNT 100`
- `#define VAL_SIZE sizeof(int)`
- `#define VAL_COUNT 100`
- `#define MEGABYTE 1000000`
- `#define GIGABYTE 1000 * MEGABYTE`
- `#define MEMORY_LIMIT 700 * MEGABYTE`

Functions

- `void * thread_func (void *arg)`
- `void set_memory_limit (size_t limit_bytes)`
- `int main ()`

Variables

- `sem_t turnstile`

4.3.1 Macro Definition Documentation

4.3.1.1 GIGABYTE

```
#define GIGABYTE 1000 * MEGABYTE
```

Definition at line 17 of file [main2.c](#).

4.3.1.2 KEY_COUNT

```
#define KEY_COUNT 100
```

Definition at line 12 of file [main2.c](#).

4.3.1.3 KEY_SIZE

```
#define KEY_SIZE 12
```

Definition at line 11 of file [main2.c](#).

4.3.1.4 MEGABYTE

```
#define MEGABYTE 1000000
```

Definition at line 16 of file [main2.c](#).

4.3.1.5 MEMORY_LIMIT

```
#define MEMORY_LIMIT 700 * MEGABYTE
```

Definition at line 18 of file [main2.c](#).

4.3.1.6 VAL_COUNT

```
#define VAL_COUNT 100
```

Definition at line 14 of file [main2.c](#).

4.3.1.7 VAL_SIZE

```
#define VAL_SIZE sizeof(int)
```

Definition at line 13 of file [main2.c](#).

4.3.2 Function Documentation

4.3.2.1 main()

```
int main ( )
```

Definition at line 135 of file [main2.c](#).

```
00135     {
00136
00137         setbuf(stdout, NULL);
00138
00139         // set_memory_limit(MEMORY_LIMIT);
00140
00141         sem_init(&turnstile, 0, 1);
00142         sem_wait(&turnstile);
00143
00144         // Create the cache with a larger size to handle more entries
00145         Cache cache = cache_create(kr_hash);
00146         if (!cache) {
00147             PRINT("Failed to create cache");
00148             return 1;
00149         }
00150
00151         const int NUM_THREADS = 1;
00152         pthread_t threads[NUM_THREADS];
00153         ThreadArgs thread_args[NUM_THREADS];
00154
00155         // Create threads
00156         for (int i = 0; i < NUM_THREADS; i++) {
00157             thread_args[i].thread_id = i;
00158             thread_args[i].cache = cache;
00159
00160             int result = pthread_create(&threads[i], NULL, thread_func, &thread_args[i]);
00161             if (result != 0) {
00162                 PRINT("Failed to create thread %d", i);
00163                 return 1;
00164             }
00165         }
00166
00167         sem_post(&turnstile);
00168
00169         // Wait for all threads to finish
00170         for (int i = 0; i < NUM_THREADS; i++) {
00171             pthread_join(threads[i], NULL);
00172         }
00173
00174         // Show cache statistics
00175         cache_stats(cache);
00176
00177         // And destroy it.
00178         cache_destroy(cache);
00179
00180         return 0;
00181     }
```

4.3.2.2 set_memory_limit()

```
void set_memory_limit (
    size_t limit_bytes )
```

Definition at line 123 of file [main2.c](#).

```
00123                                     {
00124     struct rlimit limit;
00125     limit.rlim_cur = limit_bytes; // Soft limit
00126     limit.rlim_max = limit_bytes; // Hard limit
00127
00128     if (setrlimit(RLIMIT_AS, &limit) != 0) {
00129         perror("setrlimit failed");
00130         exit(EXIT_FAILURE);
00131     }
00132 }
```

4.3.2.3 thread_func()

```
void * thread_func (
    void * arg )
```

Definition at line 30 of file [main2.c](#).

```
00030                                     {
00031
00032     // Initial turnstile so it does not run out of memory before creating threads.
00033     sem_wait(&turnstile);
00034     sem_post(&turnstile);
00035
00036     ThreadArgs* args = (ThreadArgs*)arg;
00037     int thread_id = args->thread_id;
00038     Cache cache = args->cache;
00039
00040     char* keys[KEY_COUNT];
00041     char* keysGet[KEY_COUNT];
00042     int* vals[VAL_COUNT];
00043
00044     // Generate unique keys and values for this thread
00045     for (int i = 0; i < KEY_COUNT; i++) {
00046
00047         keys[i] = dynalloc(KEY_SIZE, cache);
00048         if (!keys[i]) {
00049             PRINT("Thread %d: failed to allocate memory for key %d", thread_id, i);
00050             return NULL;
00051         }
00052
00053         keysGet[i] = dynalloc(KEY_SIZE, cache);
00054         if (!keysGet[i]) {
00055             PRINT("Thread %d: failed to allocate memory for key %d", thread_id, i);
00056             return NULL;
00057         }
00058
00059         vals[i] = dynalloc(VAL_SIZE, cache);
00060         if (!vals[i]) {
00061             PRINT("Thread %d: failed to allocate memory for value %d", thread_id, i);
00062             free(keys[i]); // Free the key if value allocation fails
00063             return NULL;
00064         }
00065
00066         // Generate a unique key for this thread
00067         snprintf(keys[i], KEY_SIZE, "key_%d_%d", thread_id, i);
00068         snprintf(keysGet[i], KEY_SIZE, "key_%d_%d", thread_id, i);
00069         *vals[i] = thread_id * KEY_COUNT + i; // Unique value for each key
00070
00071
00072         // ! Hago strlen sobre keysGet[i] porque puede que para cuando haga ese strlen ya me hayan
00073         tenido que liberar el keys[i].
00074         int result = cache_put(keys[i], strlen(keysGet[i]), vals[i], VAL_SIZE, cache);
00075         PRINT("Inserte el par clave valor numero %i.", i);
00076         if (result != 0) {
00077             PRINT("Thread %d: failed to insert key-value pair: %s / %i", thread_id, keys[i],
00078                 *vals[i]);
00079         }
00080
00081         if (i == KEY_COUNT / 2)
00082             cache_stats(cache);
00083     }
```

```

00083
00084
00085     cache_stats(cache);
00086
00087
00088
00089     // Retrieve keys from the cache
00090     for (int i = 0; i < KEY_COUNT; i++) {
00091         LookupResult lr = cache_get(keysGet[i], strlen(keysGet[i]), cache);
00092
00093         /*
00094         if (lookup_result_is_ok(lr)) {
00095             int retrieved_value = *((int*)lookup_result_get_value(lr));
00096
00097             // ! Esto puede dar segfault si ya se libero vals[i].
00098             if (retrieved_value != *vals[i]) {
00099                 PRINT("Thread %d: GET mismatch for key %s: expected %i, got %i", thread_id, keys[i],
00100                     *vals[i], retrieved_value);
00101             }
00102
00103             // PRINT("Thread %d: got the pair %s / %i", thread_id, keys[i], retrieved_value);
00104
00105         } else {
00106             PRINT("Thread %d: Failed to GET the key %s", thread_id, keys[i]);
00107         } */
00108
00109
00110     // Destroy all entries.
00111     /*
00112     for (int i = 0; i < KEY_COUNT; i++) {
00113         int result = cache_delete(keys[i], strlen(keys[i]), cache);
00114         if (result < 0) {
00115             PRINT("Thread %d: failed to delete key: %s", thread_id, keys[i]);
00116         }
00117     }
00118     */
00119
00120     return NULL;
00121 }

```

4.3.3 Variable Documentation

4.3.3.1 turnstile

sem_t turnstile

Definition at line 27 of file [main2.c](#).

4.4 main2.c

[Go to the documentation of this file.](#)

```

00001 #include <stdio.h>
00002 #include <stdlib.h>
00003 #include <string.h>
00004 #include <pthread.h>
00005 #include <time.h>
00006 #include <sys/resource.h>
00007 #include <semaphore.h>
00008
00009 #include "cache/cache.h"
00010
00011 #define KEY_SIZE 12
00012 #define KEY_COUNT 100
00013 #define VAL_SIZE sizeof(int)
00014 #define VAL_COUNT 100
00015
00016 #define MEGABYTE 1000000
00017 #define GIGABYTE 1000 * MEGABYTE
00018 #define MEMORY_LIMIT 700 * MEGABYTE
00019
00020 // Struct to pass multiple arguments to the thread function
00021 typedef struct {

```

```

00022     int thread_id;
00023     Cache cache; // Cache is already a pointer (struct Cache*)
00024 } ThreadArgs;
00025
00026
00027 sem_t turnstile;
00028
00029 // Thread function to perform cache operations
00030 void* thread_func(void* arg) {
00031
00032     // Initial turnstile so it does not run out of memory before creating threads.
00033     sem_wait(&turnstile);
00034     sem_post(&turnstile);
00035
00036     ThreadArgs* args = (ThreadArgs*)arg;
00037     int thread_id = args->thread_id;
00038     Cache cache = args->cache;
00039
00040     char* keys[KEY_COUNT];
00041     char* keysGet[KEY_COUNT];
00042     int* vals[VAL_COUNT];
00043
00044     // Generate unique keys and values for this thread
00045     for (int i = 0; i < KEY_COUNT; i++) {
00046
00047         keys[i] = dynalloc(KEY_SIZE, cache);
00048         if (!keys[i]) {
00049             PRINT("Thread %d: failed to allocate memory for key %d", thread_id, i);
00050             return NULL;
00051         }
00052
00053         keysGet[i] = dynalloc(KEY_SIZE, cache);
00054         if (!keysGet[i]) {
00055             PRINT("Thread %d: failed to allocate memory for key %d", thread_id, i);
00056             return NULL;
00057         }
00058
00059         vals[i] = dynalloc(VAL_SIZE, cache);
00060         if (!vals[i]) {
00061             PRINT("Thread %d: failed to allocate memory for value %d", thread_id, i);
00062             free(keys[i]); // Free the key if value allocation fails
00063             return NULL;
00064         }
00065
00066         // Generate a unique key for this thread
00067         snprintf(keys[i], KEY_SIZE, "key_%d_%d", thread_id, i);
00068         snprintf(keysGet[i], KEY_SIZE, "key_%d_%d", thread_id, i);
00069         *vals[i] = thread_id * KEY_COUNT + i; // Unique value for each key
00070
00071
00072         // ! Hago strlen sobre keysGet[i] porque puede que para cuando haga ese strlen ya me hayan
        tenido que liberar el keys[i].
00073         int result = cache_put(keys[i], strlen(keysGet[i]), vals[i], VAL_SIZE, cache);
00074         PRINT("Inserte el par clave valor numero %i.", i);
00075         if (result != 0) {
00076             PRINT("Thread %d: failed to insert key-value pair: %s / %i", thread_id, keys[i],
        *vals[i]);
00077         }
00078
00079         if (i == KEY_COUNT / 2)
00080             cache_stats(cache);
00081     }
00082
00083
00084     cache_stats(cache);
00085
00086
00087
00088     // Retrieve keys from the cache
00089     for (int i = 0; i < KEY_COUNT; i++) {
00090         LookupResult lr = cache_get(keysGet[i], strlen(keysGet[i]), cache);
00091
00092         /*
00093         if (lookup_result_is_ok(lr)) {
00094             int retrieved_value = *((int*)lookup_result_get_value(lr));
00095
00096             // ! Esto puede dar segfault si ya se libero vals[i].
00097             if (retrieved_value != *vals[i]) {
00098                 PRINT("Thread %d: GET mismatch for key %s: expected %i, got %i", thread_id, keys[i],
        *vals[i], retrieved_value);
00099             }
00100
00101             // PRINT("Thread %d: got the pair %s / %i", thread_id, keys[i], retrieved_value);
00102
00103         } else {
00104             PRINT("Thread %d: Failed to GET the key %s", thread_id, keys[i]);
00105         }

```

```

00106         } */
00107     }
00108
00109
00110     // Destroy all entries.
00111     /*
00112     for (int i = 0; i < KEY_COUNT; i++) {
00113         int result = cache_delete(keys[i], strlen(keys[i]), cache);
00114         if (result < 0) {
00115             PRINT("Thread %d: failed to delete key: %s", thread_id, keys[i]);
00116         }
00117     }
00118     */
00119
00120     return NULL;
00121 }
00122
00123 void set_memory_limit(size_t limit_bytes) {
00124     struct rlimit limit;
00125     limit.rlim_cur = limit_bytes; // Soft limit
00126     limit.rlim_max = limit_bytes; // Hard limit
00127
00128     if (setrlimit(RLIMIT_AS, &limit) != 0) {
00129         perror("setrlimit failed");
00130         exit(EXIT_FAILURE);
00131     }
00132 }
00133
00134
00135 int main() {
00136     setbuf(stdout, NULL);
00137
00138     // set_memory_limit(MEMORY_LIMIT);
00139
00140     sem_init(&turnstile, 0, 1);
00141     sem_wait(&turnstile);
00142
00143     // Create the cache with a larger size to handle more entries
00144     Cache cache = cache_create(kr_hash);
00145     if (!cache) {
00146         PRINT("Failed to create cache");
00147         return 1;
00148     }
00149
00150     const int NUM_THREADS = 1;
00151     pthread_t threads[NUM_THREADS];
00152     ThreadArgs thread_args[NUM_THREADS];
00153
00154     // Create threads
00155     for (int i = 0; i < NUM_THREADS; i++) {
00156         thread_args[i].thread_id = i;
00157         thread_args[i].cache = cache;
00158
00159         int result = pthread_create(&threads[i], NULL, thread_func, &thread_args[i]);
00160         if (result != 0) {
00161             PRINT("Failed to create thread %d", i);
00162             return 1;
00163         }
00164     }
00165
00166     sem_post(&turnstile);
00167
00168     // Wait for all threads to finish
00169     for (int i = 0; i < NUM_THREADS; i++) {
00170         pthread_join(threads[i], NULL);
00171     }
00172
00173     // Show cache statistics
00174     cache_stats(cache);
00175
00176     // And destroy it.
00177     cache_destroy(cache);
00178
00179     return 0;
00180 }
00181 }

```

4.5 app/main3.c File Reference

```

#include <stdlib.h>
#include <stdio.h>

```

```
#include <string.h>
#include "cache/cache.h"
```

Macros

- `#define KEY_SIZE 5`
- `#define KEY_COUNT 5`
- `#define VAL_SIZE 5`
- `#define VAL_COUNT 5`

Functions

- `int main ()`
- `int main2 ()`

4.5.1 Macro Definition Documentation

4.5.1.1 KEY_COUNT

```
#define KEY_COUNT 5
```

Definition at line 7 of file [main3.c](#).

4.5.1.2 KEY_SIZE

```
#define KEY_SIZE 5
```

Definition at line 6 of file [main3.c](#).

4.5.1.3 VAL_COUNT

```
#define VAL_COUNT 5
```

Definition at line 9 of file [main3.c](#).

4.5.1.4 VAL_SIZE

```
#define VAL_SIZE 5
```

Definition at line 8 of file [main3.c](#).

4.5.2 Function Documentation

4.5.2.1 main()

```
int main ( )
```

Definition at line 11 of file [main3.c](#).

```
00011     {
00012
00013         setbuf(stdout, NULL);
00014
00015         // Creamos la cache
00016         Cache cache = cache_create(kr_hash );
00017
00018         // Definimos los pares key-value
00019         char* keys[KEY_COUNT];
00020         int* vals[VAL_COUNT];
00021
00022
00023         const char* names[] = {
00024             "octa",
00025             "juli",
00026             "tito",
00027             "busa",
00028             "axel"
00029         };
00030
00031
00032         int nums[] = { 1, 2, 3, 4, 5 };
00033
00034         for (int i = 0; i < KEY_COUNT; i++) {
00035
00036             keys[i] = malloc(KEY_SIZE);
00037             vals[i] = malloc(sizeof(int));
00038
00039             strncpy(keys[i], names[i], KEY_SIZE);
00040             keys[i][KEY_SIZE - 1] = '\0';
00041
00042             *vals[i] = nums[i];
00043
00044             PRINT("inserted key-value pair: %s / %i", keys[i], *vals[i]);
00045
00046         }
00047
00048         // Insertamos
00049         for (int i = 0; i < KEY_COUNT; i++)
00050             PRINT("iteration %i | cache_put return: %i", i, cache_put(keys[i], KEY_SIZE, vals[i],
00051 VAL_SIZE, cache));
00052
00053         // Recuperamos las claves
00054         for (int i = 0; i < KEY_COUNT; i++) {
00055
00056             LookupResult lr = cache_get(keys[i], KEY_SIZE, cache);
00057
00058             if (lookup_result_is_ok(lr))
00059                 PRINT("GET: (%s, %i)", keys[i], *((int*) lookup_result_get_value(lr)));
00060             else
00061                 PRINT("Failed to GET the key %s", keys[i]);
00062
00063         }
00064
00065         // Eliminemos las que estan en posiciones pares
00066         PRINT("Comenzando la eliminacion.");
00067         for (int i = 0; i < KEY_COUNT; i++)
00068             if (i % 2 == 0)
00069                 cache_delete(keys[i], KEY_SIZE, cache);
00070
00071         // Recuperamos las claves nuevamente.
00072         for (int i = 0; i < KEY_COUNT; i++) {
00073
00074             LookupResult lr = cache_get(keys[i], KEY_SIZE, cache);
00075
00076             if (lookup_result_is_ok(lr))
00077                 PRINT("GET: (%s, %i)", keys[i], *((int*) lookup_result_get_value(lr)));
00078             else
00079                 PRINT("Failed to GET the key at address %p", keys[i]);
00080
00081         }
00082
00083         // Mostramos estadisticas.
00084         cache_stats(cache);
00085
00086         return 0;
00087     }
```

4.5.2.2 main2()

```
int main2 ( )
```

Definition at line 87 of file [main3.c](#).

```
00087     {
00088
00089         setbuf(stdout, NULL);
00090
00091         // Creamos la cache
00092         Cache cache = cache_create(kr_hash);
00093
00094         char name[] = "octavio";
00095         char* key = malloc(sizeof(name));
00096         strcpy(key, name);
00097
00098         int* val = malloc(sizeof(int));
00099         *val = 10;
00100
00101         // Insertamos
00102         cache_put(key, sizeof(key), val, sizeof(val), cache);
00103
00104         // Getteamos
00105         LookupResult lr = cache_get(key, KEY_SIZE, cache);
00106
00107         if (lookup_result_is_ok(lr))
00108             PRINT("GET exitoso: key %s | val %i", key, *val);
00109         else
00110             PRINT("GET fallo.");
00111
00112
00113         return 0;
00114
00115     }
```

4.6 main3.c

[Go to the documentation of this file.](#)

```
00001 #include <stdlib.h>
00002 #include <stdio.h>
00003 #include <string.h>
00004 #include "cache/cache.h"
00005
00006 #define KEY_SIZE 5
00007 #define KEY_COUNT 5
00008 #define VAL_SIZE 5
00009 #define VAL_COUNT 5
00010
00011 int main() {
00012
00013     setbuf(stdout, NULL);
00014
00015     // Creamos la cache
00016     Cache cache = cache_create(kr_hash );
00017
00018     // Definimos los pares key-value
00019     char* keys[KEY_COUNT];
00020     int* vals[VAL_COUNT];
00021
00022
00023     const char* names[] = {
00024         "octa",
00025         "juli",
00026         "tito",
00027         "busa",
00028         "axel"
00029     };
00030
00031
00032     int nums[] = { 1, 2, 3, 4, 5 };
00033
00034     for (int i = 0; i < KEY_COUNT; i++) {
00035
00036         keys[i] = malloc(KEY_SIZE);
00037         vals[i] = malloc(sizeof(int));
00038
00039         strncpy(keys[i], names[i], KEY_SIZE);
00040         keys[i][KEY_SIZE - 1] = '\0';
```



```

00041
00042     *vals[i] = nums[i];
00043
00044     PRINT("inserted key-value pair: %s / %i", keys[i], *vals[i]);
00045
00046 }
00047
00048 // Insertamos
00049 for (int i = 0; i < KEY_COUNT; i++)
00050     PRINT("iteration %i | cache_put return: %i", i, cache_put(keys[i], KEY_SIZE, vals[i],
VAL_SIZE, cache));
00051
00052 // Recuperamos las claves
00053 for (int i = 0; i < KEY_COUNT; i++) {
00054
00055     LookupResult lr = cache_get(keys[i], KEY_SIZE, cache);
00056
00057     if (lookup_result_is_ok(lr))
00058         PRINT("GET: (%s, %i)", keys[i], *((int*) lookup_result_get_value(lr)));
00059     else
00060         PRINT("Failed to GET the key %s", keys[i]);
00061 }
00062
00063 // Eliminemos las que estan en posiciones pares
00064 PRINT("Comenzando la eliminacion.");
00065 for (int i = 0; i < KEY_COUNT; i++)
00066     if (i % 2 == 0)
00067         cache_delete(keys[i], KEY_SIZE, cache);
00068
00069 // Recuperamos las claves nuevamente.
00070 for (int i = 0; i < KEY_COUNT; i++) {
00071
00072     LookupResult lr = cache_get(keys[i], KEY_SIZE, cache);
00073
00074     if (lookup_result_is_ok(lr))
00075         PRINT("GET: (%s, %i)", keys[i], *((int*) lookup_result_get_value(lr)));
00076     else
00077         PRINT("Failed to GET the key at address %p", keys[i]);
00078 }
00079
00080 // Mostramos estadisticas.
00081 cache_stats(cache);
00082
00083 return 0;
00084 }
00085
00086
00087 int main2() {
00088
00089     setbuf(stdout, NULL);
00090
00091     // Creamos la cache
00092     Cache cache = cache_create(kr_hash);
00093
00094     char name[] = "octavio";
00095     char* key = malloc(sizeof(name));
00096     strcpy(key, name);
00097
00098     int* val = malloc(sizeof(int));
00099     *val = 10;
00100
00101     // Insertamos
00102     cache_put(key, sizeof(key), val, sizeof(val), cache);
00103
00104     // Getteamos
00105     LookupResult lr = cache_get(key, KEY_SIZE, cache);
00106
00107     if (lookup_result_is_ok(lr))
00108         PRINT("GET exitoso: key %s | val %i", key, *val);
00109     else
00110         PRINT("GET fallo.");
00111
00112     return 0;
00113
00114 }
00115 }

```

4.7 cache/cache.c File Reference

```

#include <stdlib.h>
#include <pthread.h>

```

```
#include <string.h>
#include "cache.h"
#include "../hashmap/hash.h"
#include "../hashmap/hashnode.h"
#include "../lru/lru.h"
#include "../lru/lrunode.h"
#include <sys/types.h>
```

Classes

- struct [Cache](#)

Macros

- #define [BUCKETS_FACTOR](#) 100
- #define [ZONES_FACTOR](#) 10

Functions

- [Cache](#) [cache_create](#) ([HashFunction](#) hash, int num_threads)
Inicializa una [Cache](#) con funcion de hash hash.
- [LookupResult](#) [cache_get](#) (void *key, size_t key_size, [Cache](#) cache)
Busca el valor asociado a una clave en la cache.
- int [cache_put](#) (void *key, size_t key_size, void *val, size_t val_size, [Cache](#) cache)
Inserta un par clave-valor en la cache. Si la clave ya estaba asociada a un valor, lo actualiza. De ser necesario, aplica la politica de desalojo preestablecida para liberar memoria.
- int [cache_delete](#) (void *key, size_t key_size, [Cache](#) cache)
Elimina el par clave-valor asociado a key en la cache objetivo.
- [StatsReport](#) [cache_report](#) ([Cache](#) cache)
Genera un reporte de estadísticas de uso de la cache objetivo.
- ssize_t [cache_free_up_memory](#) ([Cache](#) cache, size_t memory_goal)
Libera memoria en la cache eliminando nodos. Comienza desde los que no fueron accedidos recientemente. Libera hasta un maximo de memoria determinado.
- [CacheStats](#) [cache_get_cstats](#) ([Cache](#) cache)
Obtiene el puntero a la estructura que almacena las estadísticas de la cache objetivo.
- void [cache_destroy](#) ([Cache](#) cache)
Libera la memoria de todas las componentes de la cache, incluida esta ultima.

4.7.1 Macro Definition Documentation

4.7.1.1 BUCKETS_FACTOR

```
#define BUCKETS_FACTOR 100
```

Definition at line 11 of file [cache.c](#).

4.7.1.2 ZONES_FACTOR

```
#define ZONES_FACTOR 10
```

Definition at line 12 of file [cache.c](#).

4.7.2 Function Documentation

4.7.2.1 cache_create()

```
Cache cache_create (
    HashFunction hash,
    int num_threads )
```

Inicializa una [Cache](#) con funcion de hash *hash*.

Parameters

<i>hash</i>	Funcion de hash a utilizar.
<i>num_threads</i>	Cantidad de threads que correran sobre la cache.

Returns

Un puntero a la cache creada.

Definition at line 42 of file [cache.c](#).

```
00042                                     {
00043
00044     Cache cache = malloc(sizeof(struct Cache));
00045     if (cache == NULL)
00046         return NULL;
00047
00048     if (hashmap_init(hash, cache, num_threads) != 0) {
00049         free(cache);
00050         return NULL;
00051     }
00052
00053     LRUQueue queue = lru_queue_create();
00054     if (queue == NULL) {
00055         hashmap_destroy(cache);
00056         free(cache);
00057         return NULL;
00058     }
00059
00060     CacheStats stats = cache_stats_create();
00061     if (stats == NULL) {
00062         hashmap_destroy(cache);
00063         lru_queue_destroy(queue);
00064         free(cache);
00065         return NULL;
00066     }
00067
00068     cache->queue = queue;
00069     cache->stats = stats;
00070
00071     return cache;
00072
00073 }
```

4.7.2.2 cache_delete()

```
int cache_delete (
    void * key,
```

```
size_t key_size,
Cache cache )
```

Elimina el par clave-valor asociado a *key* en la cache objetivo.

Parameters

<i>key</i>	La clave del par.
<i>key_size</i>	El tamaño de la clave.
<i>cache</i>	La cache objetivo.

Returns

0 si se elimina un par, 1 si no se encuentra el par, -1 si se produjo un error.

Definition at line 200 of file [cache.c](#).

```
00200                                     {
00201
00202     if (cache == NULL)
00203         return -1;
00204
00205     // Tomamos el lock asociado al nodo a eliminar
00206     unsigned int bucket_number = cache_get_bucket_number(key, key_size, cache);
00207
00208     // Obtenemos su zona
00209     pthread_rwlock_t* lock = cache_get_zone_mutex(bucket_number, cache);
00210     if (lock == NULL)
00211         return -1;
00212
00213     pthread_rwlock_wrlock(lock);
00214
00215     // Ahora con el lock, accedemos a su bucket y lo buscamos dentro de el.
00216     HashNode bucket = cache->buckets[bucket_number];
00217     HashNode node = hashnode_lookup_node(key, key_size, bucket);
00218
00219     // Independientemente del resultado, lo contamos como una operacion DEL
00220     cache_stats_del_counter_inc(cache->stats);
00221
00222     // Si la clave no pertenecia a la cache, devolvemos el lock y retornamos.
00223     if (node == NULL) {
00224         pthread_rwlock_unlock(lock);
00225         return 1;
00226     }
00227
00228     // Si la clave estaba en la cache borramos de la cola LRU
00229     lru_queue_delete_node(hashnode_get_prio(node), cache->queue);
00230
00231     // Lo desconectamos del hashmap
00232     hashnode_clean(node);
00233
00234     // Si era el unico nodo del bucket, node->next pasa a ser el bucket
00235     if (bucket == node)
00236         cache->buckets[bucket_number] = hashnode_get_next(node);
00237
00238     // Y liberamos su memoria
00239     size_t allocated_memory = hashnode_get_key_size(node) +
00240                             hashnode_get_val_size(node);
00241     hashnode_destroy(node);
00242
00243     pthread_rwlock_unlock(lock);
00244
00245     // Decrementamos la cantidad de keys y la cantidad de memoria asignada
00246     cache_stats_key_counter_dec(cache->stats);
00247     cache_stats_allocated_memory_free(cache->stats, allocated_memory);
00248
00249     return 0;
00250 }
00251 }
```

4.7.2.3 cache_destroy()

```
void cache_destroy (
    Cache cache )
```

Libera la memoria de todas las componentes de la cache, incluida esta ultima.

Parameters

<i>cache</i>	La cache a destruir.
--------------	----------------------

Definition at line 343 of file [cache.c](#).

```

00343                                     {
00344
00345     if (cache == NULL)
00346         return;
00347
00348     // Destruimos la LRUQueue
00349     lru_queue_destroy(cache->queue);
00350
00351     // Destruimos los nodos que queden en la hash
00352     hashmap_destroy(cache);
00353
00354     // Y destruimos las CacheStats
00355     cache_stats_destroy(cache->stats);
00356
00357     // Por ultimo, liberamos la cache.
00358     free(cache);
00359
00360 }
```

4.7.2.4 cache_free_up_memory()

```

ssize_t cache_free_up_memory (
    Cache cache,
    size_t memory_goal )
```

Libera memoria en la cache eliminando nodos. Comienza desde los que no fueron accedidos recientemente. Libera hasta un maximo de memoria determinado.

Parameters

<i>cache</i>	La cache donde se liberara memoria.
<i>memory_goal</i>	La cantidad de memoria objetivo a liberar.

Returns

La cantidad de memoria liberada al eliminar, que puede ser menor que el objetivo de memoria a liberar, o -1 si se produjo un error.

Definition at line 258 of file [cache.c](#).

```

00258                                     {
00259
00260     if (cache == NULL)
00261         return -1;
00262
00263     if (lru_queue_lock(cache->queue) != 0)
00264         return -1;
00265
00266     LRUNode lru_last_node = lru_queue_get_least_recent(cache->queue);
00267
00268     // Si la LRU estaba vacia, devolvemos el lock y retornamos un error, pues no deberia estar
    liberandose memoria si no hay elementos por eliminar.
00269     if (lru_last_node == NULL) {
00270         lru_queue_unlock(cache->queue);
00271         return -1;
00272     }
00273
00274     size_t freed_memory = 0;
00275     size_t total_freed_memory = 0;
00276     while (lru_last_node != NULL && total_freed_memory < memory_goal) {
00277
00278         // Obtenemos el numero de bucket del ultimo nodo
00279         unsigned int buck_num = lrunode_get_bucket_number(lru_last_node);
```

```

00280
00281 // Y obtenemos su zona
00282 pthread_rwlock_t* zone_lock = cache_get_zone_mutex(buck_num, cache);
00283
00284 // Guardamos las referencias a su siguiente y a su hashnode asociado pues, si obtenemos el lock de
este nodo, lo destruiremos y las perderemos.
00285 HashNode hashnode = lrunode_get_hash_node(lru_last_node);
00286 LRUNode next_node = lrunode_get_next(lru_last_node);
00287
00288 // Si el zone_lock es NULL, se produjo algun error al pedirlo.
00289 if (zone_lock == NULL) {
00290     lru_last_node = next_node;
00291     continue;
00292 }
00293
00294 // Intentamos lockear la zona
00295 if (pthread_rwlock_trywrlock(zone_lock) == 0) {
00296
00297     // Si logramos tomar el lock, eliminamos al LRUNode de la LRUQueue
lru_queue_node_clean(lru_last_node, cache->queue);
00299     lrunode_destroy(lru_last_node);
00300
00301     // Y lo eliminamos del hashmap.
HashNode bucket = cache->buckets[buck_num];
00303     if (bucket == hashnode)
00304         cache->buckets[buck_num] = hashnode_get_next(hashnode);
00305
00306     hashnode_clean(hashnode);
00307
00308     freed_memory = hashnode_get_key_size(hashnode) +
00309                   hashnode_get_val_size(hashnode);
00310
00311     total_freed_memory += freed_memory;
00312
00313     hashnode_destroy(hashnode);
00314
00315     // Soltamos el lock
pthread_rwlock_unlock(zone_lock);
00317
00318     // Y actualizamos las estadísticas
cache_stats_allocated_memory_free(cache->stats, freed_memory);
00320     cache_stats_evict_counter_inc(cache->stats);
00321     cache_stats_key_counter_dec(cache->stats);
00322
00323 }
00324
00325 lru_last_node = next_node;
00326
00327 }
00328
00329 lru_queue_unlock(cache->queue);
00330
00331 return freed_memory;
00332
00333 }

```

4.7.2.5 cache_get()

```

LookupResult cache_get (
    void * key,
    size_t key_size,
    Cache cache )

```

Busca el valor asociado a una clave en la cache.

Parameters

<i>key</i>	La clave buscada.
<i>key_size</i>	El tamaño de la clave.
<i>cache</i>	La cache objetivo.

Returns

Un LookUpResult con status indicando si la operacion fue exitosa.

Definition at line 76 of file [cache.c](#).

```

00076                                     {
00077
00078     if (cache == NULL)
00079         return create_error_lookup_result();
00080
00081     // Calculamos el bucket
00082     unsigned int bucket_number = cache_get_bucket_number(key, key_size, cache);
00083
00084     // Obtenemos el lock asociado junto con su bucket
00085     pthread_rwlock_t* lock = cache_get_zone_mutex(bucket_number, cache);
00086     if (lock == NULL)
00087         return create_error_lookup_result();
00088
00089     pthread_rwlock_rdlock(lock);
00090
00091     HashNode bucket = cache->buckets[bucket_number];
00092
00093     // Buscamos el nodo asociado a la key en el bucket
00094     HashNode node = hashnode_lookup_node(key, key_size, bucket);
00095
00096     // Si no lo encontramos, devolvemos un miss.
00097     if (node == NULL) {
00098         pthread_rwlock_unlock(lock);
00099         return create_miss_lookup_result();
00100     }
00101
00102     // Si lo encontramos, actualizamos la prioridad, devolvemos el lock,
00103     // y retornamos el valor.
00104     void* val = hashnode_get_val(node);
00105     size_t size = hashnode_get_val_size(node);
00106
00107     lru_queue_set_most_recent(hashnode_get_prio(node), cache->queue);
00108
00109     pthread_rwlock_unlock(lock);
00110
00111     cache_stats_get_counter_inc(cache->stats);
00112
00113     return create_ok_lookup_result(val, size);
00114
00115 }
```

4.7.2.6 cache_get_cstats()

```

CacheStats cache_get_cstats (
    Cache cache )
```

Obtiene el puntero a la estructura que almacena las estadísticas de la cache objetivo.

Parameters

<i>cache</i>	Cache objetivo.
--------------	---------------------------------

Returns

El puntero a la estructura [CacheStats](#).

Definition at line 336 of file [cache.c](#).

```

00336                                     {
00337     if (cache != NULL)
00338         return cache->stats;
00339     return NULL;
00340 }
```

4.7.2.7 cache_put()

```
int cache_put (
    void * key,
    size_t key_size,
    void * val,
    size_t val_size,
    Cache cache )
```

Inserta un par clave-valor en la cache. Si la clave ya estaba asociada a un valor, lo actualiza. De ser necesario, aplica la política de desalojo preestablecida para liberar memoria.

Parameters

<i>key</i>	La clave.
<i>key_size</i>	El tamaño de la clave.
<i>val</i>	El valor.
<i>val_size</i>	El tamaño del valor.
<i>cache</i>	La cache objetivo.

Returns

0 en caso de insertar un nuevo par clave-valor exitosamente, 1 si la clave ya estaba en la cache y solo se actualizo el valor, -1 si se produjo un error.

Definition at line 118 of file [cache.c](#).

```
00118                                     {
00119
00120     if (cache == NULL)
00121         return -1;
00122
00123     // Calculamos el bucket number
00124     unsigned int bucket_number = cache_get_bucket_number(key, key_size, cache);
00125
00126     // Lockemos su zona
00127     pthread_rwlock_t* lock = cache_get_zone_mutex(bucket_number, cache);
00128     if (lock == NULL)
00129         return -1;
00130
00131     pthread_rwlock_wrlock(lock);
00132
00133     // Y accedemos al bucket
00134     HashNode bucket = cache->buckets[bucket_number];
00135
00136     // Buscamos el nodo asociado a la clave en el bucket correspondiente.
00137     HashNode node = hashnode_lookup_node(key, key_size, bucket);
00138
00139     // Lo encuentre o no, la operacion de PUT se llevo a cabo.
00140     cache_stats_put_counter_inc(cache->stats);
00141
00142     // Si la clave ya pertenecia a la cache, solo actualizamos valor y prioridad.
00143     if (node != NULL) {
00144
00145         // Setteamos el nuevo valor, liberando la memoria liberada en el proceso
00146         hashnode_set_val(node, val, val_size);
00147
00148         // Restamos la memoria del valor anterior
00149         cache_stats_allocated_memory_free(cache->stats,
00150                                         hashnode_get_val_size(node));
00151
00152         // Sumamos la memoria de la nueva clave
00153         cache_stats_allocated_memory_add(cache->stats, val_size);
00154
00155         // Y actualizamos la prioridad del nodo
00156         lru_queue_set_most_recent(hashnode_get_prio(node), cache->queue);
00157
00158         pthread_rwlock_unlock(lock);
00159
00160         return 1;
00161     }
```



```

00162     }
00163
00164     // Si la clave no estaba en la cache, la insertamos.
00165     node = hashnode_create(key, key_size, val, val_size, cache);
00166     if (node == NULL) {
00167         pthread_rwlock_unlock(lock);
00168         return -1;
00169     }
00170
00171     /* Insercion:
00172        I. El prev del bucket pasa a ser node
00173        II. El next del node pasa a ser bucket
00174        III. El bucket pasa a ser node
00175     */
00176
00177     hashnode_set_prev(bucket, node);
00178     hashnode_set_next(node, bucket);
00179     cache->buckets[bucket_number] = node;
00180
00181     // Setteamos los valores de su LRUNode asociado y lo insertamos a la LRUQueue
00182     lrunode_set_bucket_number(hashnode_get_prio(node), bucket_number);
00183     lrunode_set_hash_node(hashnode_get_prio(node), node);
00184
00185     lru_queue_set_most_recent(hashnode_get_prio(node), cache->queue);
00186
00187     pthread_rwlock_unlock(lock);
00188
00189     // Sumamos la memoria del nuevo nodo y contamos la nueva key
00190     cache_stats_allocated_memory_add(cache->stats, key_size);
00191     cache_stats_allocated_memory_add(cache->stats, val_size);
00192
00193     cache_stats_key_counter_inc(cache->stats);
00194
00195     return 0;
00196
00197 }

```

4.7.2.8 cache_report()

```

StatsReport cache_report (
    Cache cache )

```

Genera un reporte de estadísticas de uso de la cache objetivo.

Parameters

<i>cache</i>	La cache objetivo.
--------------	--------------------

Returns

Un reporte con los contadores para cada operacion realizada por la cache.

Definition at line 254 of file [cache.c](#).

```

00254     {
00255     return cache_stats_report (cache->stats);
00256 }

```

4.8 cache.c

[Go to the documentation of this file.](#)

```

00001 #include <stdlib.h>
00002 #include <pthread.h>
00003 #include <string.h>
00004 #include "cache.h"
00005 #include "../hashmap/hash.h"
00006 #include "../hashmap/hashnode.h"
00007 #include "../lru/lru.h"

```

```

00008 #include "../lru/lrunode.h"
00009 #include <sys/types.h>
00010
00011 #define BUCKETS_FACTOR 100
00012 #define ZONES_FACTOR 10
00013
00014 struct Cache {
00015
00016     // Hash
00017     HashFunction      hash_function;
00018     HashNode*         buckets;
00019     int               num_buckets;
00020     pthread_rwlock_t** zone_locks;
00021     int               num_zones;
00022
00023     // LRUQueue
00024     LRUQueue queue;
00025
00026     // CacheStats
00027     CacheStats stats;
00028
00029 };
00030
00031
00032 /* Prototipos de las utilidades */
00033
00034 static int hashmap_init(HashFunction hash, Cache cache, int num_threads);
00035 static int hashmap_destroy(Cache cache);
00036 static unsigned int cache_get_bucket_number(void* key, size_t size, Cache cache);
00037 static pthread_rwlock_t* cache_get_zone_mutex(unsigned int bucket_number, Cache cache);
00038
00039
00040 /* Interfaz de la Cache */
00041
00042 Cache cache_create(HashFunction hash, int num_threads) {
00043
00044     Cache cache = malloc(sizeof(struct Cache));
00045     if (cache == NULL)
00046         return NULL;
00047
00048     if (hashmap_init(hash, cache, num_threads) != 0) {
00049         free(cache);
00050         return NULL;
00051     }
00052
00053     LRUQueue queue = lru_queue_create();
00054     if (queue == NULL) {
00055         hashmap_destroy(cache);
00056         free(cache);
00057         return NULL;
00058     }
00059
00060     CacheStats stats = cache_stats_create();
00061     if (stats == NULL) {
00062         hashmap_destroy(cache);
00063         lru_queue_destroy(queue);
00064         free(cache);
00065         return NULL;
00066     }
00067
00068     cache->queue = queue;
00069     cache->stats = stats;
00070
00071     return cache;
00072 }
00073
00074
00075
00076 LookupResult cache_get(void* key, size_t key_size, Cache cache) {
00077
00078     if (cache == NULL)
00079         return create_error_lookup_result();
00080
00081     // Calculamos el bucket
00082     unsigned int bucket_number = cache_get_bucket_number(key, key_size, cache);
00083
00084     // Obtenemos el lock asociado junto con su bucket
00085     pthread_rwlock_t* lock = cache_get_zone_mutex(bucket_number, cache);
00086     if (lock == NULL)
00087         return create_error_lookup_result();
00088
00089     pthread_rwlock_rdlock(lock);
00090
00091     HashNode bucket = cache->buckets[bucket_number];
00092
00093     // Buscamos el nodo asociado a la key en el bucket
00094     HashNode node = hashnode_lookup_node(key, key_size, bucket);

```

```

00095
00096 // Si no lo encontramos, devolvemos un miss.
00097 if (node == NULL) {
00098     pthread_rwlock_unlock(lock);
00099     return create_miss_lookup_result();
00100 }
00101
00102 // Si lo encontramos, actualizamos la prioridad, devolvemos el lock,
00103 // y retornamos el valor.
00104 void* val = hashnode_get_val(node);
00105 size_t size = hashnode_get_val_size(node);
00106
00107 lru_queue_set_most_recent(hashnode_get_prio(node), cache->queue);
00108
00109 pthread_rwlock_unlock(lock);
00110
00111 cache_stats_get_counter_inc(cache->stats);
00112
00113 return create_ok_lookup_result(val, size);
00114 }
00115 }
00116
00117
00118 int cache_put(void* key, size_t key_size, void* val, size_t val_size, Cache cache) {
00119
00120     if (cache == NULL)
00121         return -1;
00122
00123     // Calculamos el bucket number
00124     unsigned int bucket_number = cache_get_bucket_number(key, key_size, cache);
00125
00126     // Lockemos su zona
00127     pthread_rwlock_t* lock = cache_get_zone_mutex(bucket_number, cache);
00128     if (lock == NULL)
00129         return -1;
00130
00131     pthread_rwlock_wrlock(lock);
00132
00133     // Y accedemos al bucket
00134     HashNode bucket = cache->buckets[bucket_number];
00135
00136     // Buscamos el nodo asociado a la clave en el bucket correspondiente.
00137     HashNode node = hashnode_lookup_node(key, key_size, bucket);
00138
00139     // Lo encuentre o no, la operacion de PUT se llevo a cabo.
00140     cache_stats_put_counter_inc(cache->stats);
00141
00142     // Si la clave ya pertenecia a la cache, solo actualizamos valor y prioridad.
00143     if (node != NULL) {
00144
00145         // Setteamos el nuevo valor, liberando la memoria liberada en el proceso
00146         hashnode_set_val(node, val, val_size);
00147
00148         // Restamos la memoria del valor anterior
00149         cache_stats_allocated_memory_free(cache->stats,
00150             hashnode_get_val_size(node));
00151
00152         // Sumamos la memoria de la nueva clave
00153         cache_stats_allocated_memory_add(cache->stats, val_size);
00154
00155         // Y actualizamos la prioridad del nodo
00156         lru_queue_set_most_recent(hashnode_get_prio(node), cache->queue);
00157
00158         pthread_rwlock_unlock(lock);
00159
00160         return 1;
00161     }
00162
00163     // Si la clave no estaba en la cache, la insertamos.
00164     node = hashnode_create(key, key_size, val, val_size, cache);
00165     if (node == NULL) {
00166         pthread_rwlock_unlock(lock);
00167         return -1;
00168     }
00169
00170     /* Insercion:
00171     I. El prev del bucket pasa a ser node
00172     II. El next del node pasa a ser bucket
00173     III. El bucket pasa a ser node
00174     */
00175
00176     hashnode_set_prev(bucket, node);
00177     hashnode_set_next(node, bucket);
00178     cache->buckets[bucket_number] = node;
00179
00180     // Setteamos los valores de su LRUNode asociado y lo insertamos a la LRUQueue

```

```

00182     lruqueue_set_bucket_number(hashnode_get_prio(node), bucket_number);
00183     lruqueue_set_hash_node(hashnode_get_prio(node), node);
00184
00185     lruqueue_set_most_recent(hashnode_get_prio(node), cache->queue);
00186
00187     pthread_rwlock_unlock(lock);
00188
00189     // Sumamos la memoria del nuevo nodo y contamos la nueva key
00190     cache_stats_allocated_memory_add(cache->stats, key_size);
00191     cache_stats_allocated_memory_add(cache->stats, val_size);
00192
00193     cache_stats_key_counter_inc(cache->stats);
00194
00195     return 0;
00196
00197 }
00198
00199 int cache_delete(void* key, size_t key_size, Cache cache) {
00200     if (cache == NULL)
00201         return -1;
00202
00203     // Tomamos el lock asociado al nodo a eliminar
00204     unsigned int bucket_number = cache_get_bucket_number(key, key_size, cache);
00205
00206     // Obtenemos su zona
00207     pthread_rwlock_t* lock = cache_get_zone_mutex(bucket_number, cache);
00208     if (lock == NULL)
00209         return -1;
00210
00211     pthread_rwlock_wrlock(lock);
00212
00213     // Ahora con el lock, accedemos a su bucket y lo buscamos dentro de el.
00214     HashNode bucket = cache->buckets[bucket_number];
00215     HashNode node = hashnode_lookup_node(key, key_size, bucket);
00216
00217     // Independientemente del resultado, lo contamos como una operacion DEL
00218     cache_stats_del_counter_inc(cache->stats);
00219
00220     // Si la clave no pertenecia a la cache, devolvemos el lock y retornamos.
00221     if (node == NULL) {
00222         pthread_rwlock_unlock(lock);
00223         return 1;
00224     }
00225
00226     // Si la clave estaba en la cache borramos de la cola LRU
00227     lruqueue_delete_node(hashnode_get_prio(node), cache->queue);
00228
00229     // Lo desconectamos del hashmap
00230     hashnode_clean(node);
00231
00232     // Si era el unico nodo del bucket, node->next pasa a ser el bucket
00233     if (bucket == node)
00234         cache->buckets[bucket_number] = hashnode_get_next(node);
00235
00236     // Y liberamos su memoria
00237     size_t allocated_memory = hashnode_get_key_size(node) +
00238                             hashnode_get_val_size(node);
00239     hashnode_destroy(node);
00240
00241     pthread_rwlock_unlock(lock);
00242
00243     // Decrementamos la cantidad de keys y la cantidad de memoria asignada
00244     cache_stats_key_counter_dec(cache->stats);
00245     cache_stats_allocated_memory_free(cache->stats, allocated_memory);
00246
00247     return 0;
00248
00249 }
00250
00251 StatsReport cache_report(Cache cache) {
00252     return cache_stats_report(cache->stats);
00253 }
00254
00255 ssize_t cache_free_up_memory(Cache cache, size_t memory_goal) {
00256     if (cache == NULL)
00257         return -1;
00258
00259     if (lruqueue_lock(cache->queue) != 0)
00260         return -1;
00261
00262     LRUNode lru_last_node = lruqueue_get_least_recent(cache->queue);
00263
00264     // Si la LRU estaba vacia, devolvemos el lock y retornamos un error, pues no deberia estar

```

```

    liberandose memoria si no hay elementos por eliminar.
00269     if (lru_last_node == NULL) {
00270         lru_queue_unlock(cache->queue);
00271         return -1;
00272     }
00273
00274     size_t freed_memory = 0;
00275     size_t total_freed_memory = 0;
00276     while (lru_last_node != NULL && total_freed_memory < memory_goal) {
00277         // Obtenemos el numero de bucket del ultimo nodo
00278         unsigned int buck_num = lrunode_get_bucket_number(lru_last_node);
00279
00280         // Y obtenemos su zona
00281         pthread_rwlock_t* zone_lock = cache_get_zone_mutex(buck_num, cache);
00282
00283         // Guardamos las referencias a su siguiente y a su hashnode asociado pues, si obtenemos el lock de
este nodo, lo destruiremos y las perderemos.
00285         HashNode hashnode = lrunode_get_hash_node(lru_last_node);
00286         LRUNode next_node = lrunode_get_next(lru_last_node);
00287
00288         // Si el zone_lock es NULL, se produjo algun error al pedirlo.
00289         if (zone_lock == NULL) {
00290             lru_last_node = next_node;
00291             continue;
00292         }
00293
00294         // Intentamos lockear la zona
00295         if (pthread_rwlock_trywrlock(zone_lock) == 0) {
00296
00297             // Si logramos tomar el lock, eliminamos al LRUNode de la LRUQueue
00298             lru_queue_node_clean(lru_last_node, cache->queue);
00299             lrunode_destroy(lru_last_node);
00300
00301             // Y lo eliminamos del hashmap.
00302             HashNode bucket = cache->buckets[buck_num];
00303             if (bucket == hashnode)
00304                 cache->buckets[buck_num] = hashnode_get_next(hashnode);
00305
00306             hashnode_clean(hashnode);
00307
00308             freed_memory = hashnode_get_key_size(hashnode) +
00309                             hashnode_get_val_size(hashnode);
00310
00311             total_freed_memory += freed_memory;
00312
00313             hashnode_destroy(hashnode);
00314
00315             // Soltamos el lock
00316             pthread_rwlock_unlock(zone_lock);
00317
00318             // Y actualizamos las estadisticas
00319             cache_stats_allocated_memory_free(cache->stats, freed_memory);
00320             cache_stats_evict_counter_inc(cache->stats);
00321             cache_stats_key_counter_dec(cache->stats);
00322         }
00323     }
00324     lru_last_node = next_node;
00325 }
00326
00327 lru_queue_unlock(cache->queue);
00328
00329 return freed_memory;
00330 }
00331
00332
00333 }
00334
00335
00336 CacheStats cache_get_cstats(Cache cache) {
00337     if (cache != NULL)
00338         return cache->stats;
00339     return NULL;
00340 }
00341
00342
00343 void cache_destroy(Cache cache) {
00344     if (cache == NULL)
00345         return;
00346
00347     // Destruimos la LRUQueue
00348     lru_queue_destroy(cache->queue);
00349
00350     // Destruimos los nodos que queden en la hash
00351     hashmap_destroy(cache);
00352
00353

```

```

00354 // Y destruimos las CacheStats
00355 cache_stats_destroy(cache->stats);
00356
00357 // Por ultimo, liberamos la cache.
00358 free(cache);
00359
00360 }
00361
00362
00363 /* Funciones de utilidad no exportadas */
00364
00365 /**
00366  * @brief Inicializa los campos asociados al HashMap de la cache, setteando a `hash` como funcion de
00367  * hash.
00368  * @param hash Funcion de hash a utilizar en la cache.
00369  * @param cache Puntero a la cache donde se inicializara el HashMap.
00370  * @return 0 si es exitoso, -1 en caso de producirse un error al inicializar el HashMap.
00371  */
00372 static int hashmap_init(HashFunction hash, Cache cache, int num_threads) {
00373
00374     cache->hash_function = hash;
00375     int num_zones = num_threads * ZONES_FACTOR;
00376     int num_buckets = num_zones * BUCKETS_FACTOR;
00377
00378     // Creamos el doble de zonas que de threads.
00379     cache->zone_locks = malloc(sizeof(pthread_rwlock_t) * num_zones);
00380     if (cache->zone_locks == NULL)
00381         return -1;
00382
00383     // Y la cantidad de buckets viene dada por la cantidad de zonas por un factor
00384     cache->buckets = malloc(sizeof(HashNode) * num_buckets);
00385     if (cache->buckets == NULL) {
00386         free(cache->zone_locks);
00387         return -1;
00388     }
00389
00390     memset(cache->buckets, 0, sizeof(HashNode) * num_buckets);
00391
00392
00393     // Inicializamos los locks
00394     int mutex_error = 0;
00395     for (int i = 0; i < num_zones; i++) {
00396         cache->zone_locks[i] = malloc(sizeof(pthread_rwlock_t));
00397         if (cache->zone_locks[i] == NULL) {
00398             free(cache->zone_locks);
00399             free(cache->buckets);
00400             for (int j = 0; j < i; j++) free(cache->zone_locks[j]);
00401             return -1;
00402         }
00403
00404         mutex_error = mutex_error || pthread_rwlock_init(cache->zone_locks[i], NULL);
00405     }
00406
00407     cache->num_zones = num_zones;
00408     cache->num_buckets = num_buckets;
00409
00410     if (mutex_error)
00411         return -1;
00412
00413     return 0;
00414 }
00415
00416
00417
00418 /**
00419  * @brief Destruye los campos asociados al HashMap de la cache objetivo.
00420  *
00421  * @param cache Puntero a la cache donde se destruira el HashMap.
00422  * @return 0 si es exitoso, distinto de 0 si se produce un error al liberar memoria.
00423  */
00424 static int hashmap_destroy(Cache cache) {
00425
00426     if (cache == NULL)
00427         return 1;
00428
00429     // Tomamos los locks de todas las zonas
00430     int status = 0;
00431     for (int i = 0; i < cache->num_zones; i++)
00432         status = pthread_rwlock_wrlock(cache->zone_locks[i]) || status;
00433
00434     // Y pasamos a recorrer cada bucket destruyendo todos los nodos.
00435     HashNode tmp, next;
00436
00437     for (int i = 0; i < cache->num_buckets; i++) {
00438
00439         tmp = cache->buckets[i];

```

```

00440
00441     while (tmp) {
00442
00443         next = hashnode_get_next(tmp);
00444
00445         // No importa 'limpiarlos' pues vamos a destruir a todos.
00446         hashnode_destroy(tmp);
00447
00448         tmp = next;
00449     }
00450 }
00451
00452 }
00453
00454 // Destruimos todos los pthread_rwlock_t
00455 for (int i = 0; i < cache->num_zones; i++) {
00456     pthread_rwlock_destroy(cache->zone_locks[i]);
00457     free(cache->zone_locks[i]);
00458 }
00459
00460 // Tambien destruimos los arrays de locks y arrays de buckets
00461 free(cache->zone_locks);
00462 free(cache->buckets);
00463
00464 return 0;
00465
00466 }
00467
00468 /**
00469 * @brief Calcula el numero de bucket asociado a `key` en la cache objetivo.
00470 *
00471 * @param key La clave de la cual queremos saber su numero de bucket.
00472 * @param size La longitud de la clave.
00473 * @param cache La cache objetivo.
00474 * @return El numero de bucket asociado a la key en la cache.
00475 */
00476 static unsigned int cache_get_bucket_number(void* key, size_t size, Cache cache) {
00477     return cache->hash_function(key, size) % cache->num_buckets;
00478 }
00479
00480 /**
00481 * @brief Obtiene un puntero al mutex asociado al numero de bucket ingresado en la cache objetivo.
00482 *
00483 * @param bucket_number El numero de bucket para el cual queremos obtener su mutex.
00484 * @param cache La cache objetivo.
00485 * @return Un puntero al mutex asociado al bucket, que es `NULL` en caso de producirse un error o no
00486         existir.
00487 */
00488 static pthread_rwlock_t* cache_get_zone_mutex(unsigned int bucket_number, Cache cache) {
00489     if (cache == NULL)
00490         return NULL;
00491
00492     return cache->zone_locks[bucket_number % cache->num_zones];
00493 }
00494
00495 }
00496

```

4.9 cache/cache.h File Reference

```

#include "cache_stats.h"
#include "../dynalloc/dynalloc.h"
#include "../helpers/results.h"

```

Macros

- #define `DEBUG` 1
- #define `PRINT`(fmt, ...) printf("[%s] " fmt "\n", __func__, ##__VA_ARGS__)

Typedefs

- typedef struct `Cache` * `Cache`
- typedef unsigned long(* `HashFunction`) (void *, size_t)

Functions

- unsigned long [kr_hash](#) (char *key, size_t size)
- [Cache](#) [cache_create](#) (HashFunction hash, int num_threads)
Inicializa una [Cache](#) con funcion de hash hash.
- [LookupResult](#) [cache_get](#) (void *key, size_t key_size, [Cache](#) cache)
Busca el valor asociado a una clave en la cache.
- int [cache_put](#) (void *key, size_t key_size, void *val, size_t val_size, [Cache](#) cache)
Inserta un par clave-valor en la cache. Si la clave ya estaba asociada a un valor, lo actualiza. De ser necesario, aplica la politica de desalojo preestablecida para liberar memoria.
- int [cache_delete](#) (void *key, size_t key_size, [Cache](#) cache)
Elimina el par clave-valor asociado a key en la cache objetivo.
- [StatsReport](#) [cache_report](#) ([Cache](#) cache)
Genera un reporte de estadisticas de uso de la cache objetivo.
- ssize_t [cache_free_up_memory](#) ([Cache](#) cache, size_t memory_goal)
Libera memoria en la cache eliminando nodos. Comienza desde los que no fueron accedidos recientemente. Libera hasta un maximo de memoria determinado.
- [CacheStats](#) [cache_get_cstats](#) ([Cache](#) cache)
Obtiene el puntero a la estructura que almacena las estadisticas de la cache objetivo.
- void [cache_destroy](#) ([Cache](#) cache)
Libera la memoria de todas las componentes de la cache, incluida esta ultima.

4.9.1 Macro Definition Documentation

4.9.1.1 DEBUG

```
#define DEBUG 1
```

Definition at line 10 of file [cache.h](#).

4.9.1.2 PRINT

```
#define PRINT(  
    fmt,  
    ... ) printf("[%s] " fmt "\n", __func__, ##__VA_ARGS__)
```

Definition at line 12 of file [cache.h](#).

4.9.2 Typedef Documentation

4.9.2.1 Cache

```
typedef struct Cache* Cache
```

Definition at line 22 of file [cache.h](#).

4.9.2.2 HashFunction

```
typedef unsigned long(* HashFunction) (void *, size_t)
```

Definition at line 23 of file [cache.h](#).

4.9.3 Function Documentation

4.9.3.1 cache_create()

```
Cache cache_create (
    HashFunction hash,
    int num_threads )
```

Inicializa una [Cache](#) con funcion de hash *hash*.

Parameters

<i>hash</i>	Funcion de hash a utilizar.
<i>num_threads</i>	Cantidad de threads que correran sobre la cache.

Returns

Un puntero a la cache creada.

Definition at line 42 of file [cache.c](#).

```
00042                                     {
00043
00044     Cache cache = malloc(sizeof(struct Cache));
00045     if (cache == NULL)
00046         return NULL;
00047
00048     if (hashmap_init(hash, cache, num_threads) != 0) {
00049         free(cache);
00050         return NULL;
00051     }
00052
00053     LRUQueue queue = lru_queue_create();
00054     if (queue == NULL) {
00055         hashmap_destroy(cache);
00056         free(cache);
00057         return NULL;
00058     }
00059
00060     CacheStats stats = cache_stats_create();
00061     if (stats == NULL) {
00062         hashmap_destroy(cache);
00063         lru_queue_destroy(queue);
00064         free(cache);
00065         return NULL;
00066     }
00067
00068     cache->queue = queue;
00069     cache->stats = stats;
00070
00071     return cache;
00072
00073 }
```

4.9.3.2 cache_delete()

```
int cache_delete (
    void * key,
```

```
size_t key_size,
Cache cache )
```

Elimina el par clave-valor asociado a *key* en la cache objetivo.

Parameters

<i>key</i>	La clave del par.
<i>key_size</i>	El tamaño de la clave.
<i>cache</i>	La cache objetivo.

Returns

0 si se elimina un par, 1 si no se encuentra el par, -1 si se produjo un error.

Definition at line 200 of file [cache.c](#).

```
00200                                     {
00201
00202     if (cache == NULL)
00203         return -1;
00204
00205     // Tomamos el lock asociado al nodo a eliminar
00206     unsigned int bucket_number = cache_get_bucket_number(key, key_size, cache);
00207
00208     // Obtenemos su zona
00209     pthread_rwlock_t* lock = cache_get_zone_mutex(bucket_number, cache);
00210     if (lock == NULL)
00211         return -1;
00212
00213     pthread_rwlock_wrlock(lock);
00214
00215     // Ahora con el lock, accedemos a su bucket y lo buscamos dentro de el.
00216     HashNode bucket = cache->buckets[bucket_number];
00217     HashNode node = hashnode_lookup_node(key, key_size, bucket);
00218
00219     // Independientemente del resultado, lo contamos como una operacion DEL
00220     cache_stats_del_counter_inc(cache->stats);
00221
00222     // Si la clave no pertenecia a la cache, devolvemos el lock y retornamos.
00223     if (node == NULL) {
00224         pthread_rwlock_unlock(lock);
00225         return 1;
00226     }
00227
00228     // Si la clave estaba en la cache borramos de la cola LRU
00229     lru_queue_delete_node(hashnode_get_prio(node), cache->queue);
00230
00231     // Lo desconectamos del hashmap
00232     hashnode_clean(node);
00233
00234     // Si era el unico nodo del bucket, node->next pasa a ser el bucket
00235     if (bucket == node)
00236         cache->buckets[bucket_number] = hashnode_get_next(node);
00237
00238     // Y liberamos su memoria
00239     size_t allocated_memory = hashnode_get_key_size(node) +
00240                             hashnode_get_val_size(node);
00241     hashnode_destroy(node);
00242
00243     pthread_rwlock_unlock(lock);
00244
00245     // Decrementamos la cantidad de keys y la cantidad de memoria asignada
00246     cache_stats_key_counter_dec(cache->stats);
00247     cache_stats_allocated_memory_free(cache->stats, allocated_memory);
00248
00249     return 0;
00250 }
00251 }
```

4.9.3.3 cache_destroy()

```
void cache_destroy (
    Cache cache )
```

Libera la memoria de todas las componentes de la cache, incluida esta ultima.

Parameters

<i>cache</i>	La cache a destruir.
--------------	----------------------

Definition at line 343 of file [cache.c](#).

```

00343                                     {
00344
00345     if (cache == NULL)
00346         return;
00347
00348     // Destruimos la LRUQueue
00349     lru_queue_destroy(cache->queue);
00350
00351     // Destruimos los nodos que queden en la hash
00352     hashmap_destroy(cache);
00353
00354     // Y destruimos las CacheStats
00355     cache_stats_destroy(cache->stats);
00356
00357     // Por ultimo, liberamos la cache.
00358     free(cache);
00359
00360 }
```

4.9.3.4 cache_free_up_memory()

```

ssize_t cache_free_up_memory (
    Cache cache,
    size_t memory_goal )
```

Libera memoria en la cache eliminando nodos. Comienza desde los que no fueron accedidos recientemente. Libera hasta un maximo de memoria determinado.

Parameters

<i>cache</i>	La cache donde se liberara memoria.
<i>memory_goal</i>	La cantidad de memoria objetivo a liberar.

Returns

La cantidad de memoria liberada al eliminar, que puede ser menor que el objetivo de memoria a liberar, o -1 si se produjo un error.

Definition at line 258 of file [cache.c](#).

```

00258                                     {
00259
00260     if (cache == NULL)
00261         return -1;
00262
00263     if (lru_queue_lock(cache->queue) != 0)
00264         return -1;
00265
00266     LRUNode lru_last_node = lru_queue_get_least_recent(cache->queue);
00267
00268     // Si la LRU estaba vacia, devolvemos el lock y retornamos un error, pues no deberia estar
    liberandose memoria si no hay elementos por eliminar.
00269     if (lru_last_node == NULL) {
00270         lru_queue_unlock(cache->queue);
00271         return -1;
00272     }
00273
00274     size_t freed_memory = 0;
00275     size_t total_freed_memory = 0;
00276     while (lru_last_node != NULL && total_freed_memory < memory_goal) {
00277
00278         // Obtenemos el numero de bucket del ultimo nodo
00279         unsigned int buck_num = lrunode_get_bucket_number(lru_last_node);
```

```

00280
00281 // Y obtenemos su zona
00282 pthread_rwlock_t* zone_lock = cache_get_zone_mutex(buck_num, cache);
00283
00284 // Guardamos las referencias a su siguiente y a su hashnode asociado pues, si obtenemos el lock de
este nodo, lo destruiremos y las perderemos.
00285 HashNode hashnode = lrunode_get_hash_node(lru_last_node);
00286 LRUNode next_node = lrunode_get_next(lru_last_node);
00287
00288 // Si el zone_lock es NULL, se produjo algun error al pedirlo.
00289 if (zone_lock == NULL) {
00290     lru_last_node = next_node;
00291     continue;
00292 }
00293
00294 // Intentamos lockear la zona
00295 if (pthread_rwlock_trywrlock(zone_lock) == 0) {
00296
00297     // Si logramos tomar el lock, eliminamos al LRUNode de la LRUQueue
lru_queue_node_clean(lru_last_node, cache->queue);
00299     lrunode_destroy(lru_last_node);
00300
00301     // Y lo eliminamos del hashmap.
HashNode bucket = cache->buckets[buck_num];
00303     if (bucket == hashnode)
00304         cache->buckets[buck_num] = hashnode_get_next(hashnode);
00305
00306     hashnode_clean(hashnode);
00307
00308     freed_memory = hashnode_get_key_size(hashnode) +
00309                   hashnode_get_val_size(hashnode);
00310
00311     total_freed_memory += freed_memory;
00312
00313     hashnode_destroy(hashnode);
00314
00315     // Soltamos el lock
pthread_rwlock_unlock(zone_lock);
00317
00318     // Y actualizamos las estadísticas
cache_stats_allocated_memory_free(cache->stats, freed_memory);
00320     cache_stats_evict_counter_inc(cache->stats);
00321     cache_stats_key_counter_dec(cache->stats);
00322
00323 }
00324
00325 lru_last_node = next_node;
00326
00327 }
00328
00329 lru_queue_unlock(cache->queue);
00330
00331 return freed_memory;
00332
00333 }

```

4.9.3.5 cache_get()

```

LookupResult cache_get (
    void * key,
    size_t key_size,
    Cache cache )

```

Busca el valor asociado a una clave en la cache.

Parameters

<i>key</i>	La clave buscada.
<i>key_size</i>	El tamaño de la clave.
<i>cache</i>	La cache objetivo.

Returns

Un `LookupResult` con status indicando si la operacion fue exitosa.

Definition at line 76 of file `cache.c`.

```

00076                                     {
00077
00078     if (cache == NULL)
00079         return create_error_lookup_result();
00080
00081     // Calculamos el bucket
00082     unsigned int bucket_number = cache_get_bucket_number(key, key_size, cache);
00083
00084     // Obtenemos el lock asociado junto con su bucket
00085     pthread_rwlock_t* lock = cache_get_zone_mutex(bucket_number, cache);
00086     if (lock == NULL)
00087         return create_error_lookup_result();
00088
00089     pthread_rwlock_rdlock(lock);
00090
00091     HashNode bucket = cache->buckets[bucket_number];
00092
00093     // Buscamos el nodo asociado a la key en el bucket
00094     HashNode node = hashnode_lookup_node(key, key_size, bucket);
00095
00096     // Si no lo encontramos, devolvemos un miss.
00097     if (node == NULL) {
00098         pthread_rwlock_unlock(lock);
00099         return create_miss_lookup_result();
00100     }
00101
00102     // Si lo encontramos, actualizamos la prioridad, devolvemos el lock,
00103     // y retornamos el valor.
00104     void* val = hashnode_get_val(node);
00105     size_t size = hashnode_get_val_size(node);
00106
00107     lru_queue_set_most_recent(hashnode_get_prio(node), cache->queue);
00108
00109     pthread_rwlock_unlock(lock);
00110
00111     cache_stats_get_counter_inc(cache->stats);
00112
00113     return create_ok_lookup_result(val, size);
00114
00115 }
```

4.9.3.6 cache_get_cstats()

```

CacheStats cache_get_cstats (
    Cache cache )
```

Obtiene el puntero a la estructura que almacena las estadisticas de la cache objetivo.

Parameters

<code>cache</code>	Cache objetivo.
--------------------	-----------------

Returns

El puntero a la estructura `CacheStats`.

Definition at line 336 of file `cache.c`.

```

00336                                     {
00337     if (cache != NULL)
00338         return cache->stats;
00339     return NULL;
00340 }
```

4.9.3.7 cache_put()

```
int cache_put (
    void * key,
    size_t key_size,
    void * val,
    size_t val_size,
    Cache cache )
```

Inserta un par clave-valor en la cache. Si la clave ya estaba asociada a un valor, lo actualiza. De ser necesario, aplica la política de desalojo preestablecida para liberar memoria.

Parameters

<i>key</i>	La clave.
<i>key_size</i>	El tamaño de la clave.
<i>val</i>	El valor.
<i>val_size</i>	El tamaño del valor.
<i>cache</i>	La cache objetivo.

Returns

0 en caso de insertar un nuevo par clave-valor exitosamente, 1 si la clave ya estaba en la cache y solo se actualizo el valor, -1 si se produjo un error.

Definition at line 118 of file [cache.c](#).

```
00118                                     {
00119
00120     if (cache == NULL)
00121         return -1;
00122
00123     // Calculamos el bucket number
00124     unsigned int bucket_number = cache_get_bucket_number(key, key_size, cache);
00125
00126     // Lockemos su zona
00127     pthread_rwlock_t* lock = cache_get_zone_mutex(bucket_number, cache);
00128     if (lock == NULL)
00129         return -1;
00130
00131     pthread_rwlock_wrlock(lock);
00132
00133     // Y accedemos al bucket
00134     HashNode bucket = cache->buckets[bucket_number];
00135
00136     // Buscamos el nodo asociado a la clave en el bucket correspondiente.
00137     HashNode node = hashnode_lookup_node(key, key_size, bucket);
00138
00139     // Lo encuentre o no, la operacion de PUT se llevo a cabo.
00140     cache_stats_put_counter_inc(cache->stats);
00141
00142     // Si la clave ya pertenecia a la cache, solo actualizamos valor y prioridad.
00143     if (node != NULL) {
00144
00145         // Setteamos el nuevo valor, liberando la memoria liberada en el proceso
00146         hashnode_set_val(node, val, val_size);
00147
00148         // Restamos la memoria del valor anterior
00149         cache_stats_allocated_memory_free(cache->stats,
00150                                         hashnode_get_val_size(node));
00151
00152         // Sumamos la memoria de la nueva clave
00153         cache_stats_allocated_memory_add(cache->stats, val_size);
00154
00155         // Y actualizamos la prioridad del nodo
00156         lru_queue_set_most_recent(hashnode_get_prio(node), cache->queue);
00157
00158         pthread_rwlock_unlock(lock);
00159
00160         return 1;
00161     }
```

```

00162     }
00163
00164     // Si la clave no estaba en la cache, la insertamos.
00165     node = hashnode_create(key, key_size, val, val_size, cache);
00166     if (node == NULL) {
00167         pthread_rwlock_unlock(lock);
00168         return -1;
00169     }
00170
00171     /* Insercion:
00172        I. El prev del bucket pasa a ser node
00173        II. El next del node pasa a ser bucket
00174        III. El bucket pasa a ser node
00175     */
00176
00177     hashnode_set_prev(bucket, node);
00178     hashnode_set_next(node, bucket);
00179     cache->buckets[bucket_number] = node;
00180
00181     // Setteamos los valores de su LRUNode asociado y lo insertamos a la LRUQueue
00182     lrunode_set_bucket_number(hashnode_get_prio(node), bucket_number);
00183     lrunode_set_hash_node(hashnode_get_prio(node), node);
00184
00185     lru_queue_set_most_recent(hashnode_get_prio(node), cache->queue);
00186
00187     pthread_rwlock_unlock(lock);
00188
00189     // Sumamos la memoria del nuevo nodo y contamos la nueva key
00190     cache_stats_allocated_memory_add(cache->stats, key_size);
00191     cache_stats_allocated_memory_add(cache->stats, val_size);
00192
00193     cache_stats_key_counter_inc(cache->stats);
00194
00195     return 0;
00196
00197 }

```

4.9.3.8 cache_report()

```

StatsReport cache_report (
    Cache cache )

```

Genera un reporte de estadísticas de uso de la cache objetivo.

Parameters

<i>cache</i>	La cache objetivo.
--------------	--------------------

Returns

Un reporte con los contadores para cada operacion realizada por la cache.

Definition at line 254 of file [cache.c](#).

```

00254     {
00255     return cache_stats_report(cache->stats);
00256 }

```

4.9.3.9 kr_hash()

```

unsigned long kr_hash (
    char * key,
    size_t size )

```

Definition at line 3 of file [hash.c](#).

```

00003     {
00004

```

```

00005    unsigned long hashval;
00006    unsigned long i;
00007
00008    for (i = 0, hashval = 0 ; i < size ; ++i, key++)
00009        hashval = *key + 31 * hashval;
00010
00011    return hashval;
00012 }

```

4.10 cache.h

[Go to the documentation of this file.](#)

```

00001 #ifndef __CACHE_H__
00002 #define __CACHE_H__
00003
00004 #include "cache_stats.h"
00005 #include "../dynalloc/dynalloc.h"
00006 #include "../helpers/results.h"
00007
00008
00009 // Macro para debugging global
00010 #define DEBUG 1
00011 #if DEBUG == 1
00012     #define PRINT(fmt, ...) printf("[%s] " fmt "\n", __func__, ##__VA_ARGS__)
00013 #else
00014     #define PRINT(fmt, ...) ((void)0) // Expands to nothing when DEBUG is not 1
00015 #endif
00016
00017
00018 // Hash default
00019 unsigned long kr_hash(char* key, size_t size);
00020
00021 // Forward declarations para evitar los doble include.
00022 typedef struct Cache* Cache;
00023 typedef unsigned long (*HashFunction)(void* , size_t);
00024
00025 /**
00026  * @brief Inicializa una Cache con funcion de hash \a hash.
00027  *
00028  * @param hash Funcion de hash a utilizar.
00029  * @param num_threads Cantidad de threads que correran sobre la cache.
00030  *
00031  * @return Un puntero a la cache creada.
00032  */
00033 Cache cache_create(HashFunction hash, int num_threads);
00034
00035
00036 /**
00037  * @brief Busca el valor asociado a una clave en la cache.
00038  *
00039  * @param key La clave buscada.
00040  * @param key_size El tamaño de la clave.
00041  * @param cache La cache objetivo.
00042  *
00043  * @return Un LookupResult con status indicando si la operacion fue exitosa.
00044  */
00045 LookupResult cache_get(void* key, size_t key_size, Cache cache);
00046
00047
00048 /**
00049  * @brief Inserta un par clave-valor en la cache. Si la clave ya estaba
00050  * asociada a un valor, lo actualiza. De ser necesario, aplica la politica
00051  * de desalojo preestablecida para liberar memoria.
00052  *
00053  * @param key La clave.
00054  * @param key_size El tamaño de la clave.
00055  * @param val El valor.
00056  * @param val_size El tamaño del valor.
00057  * @param cache La cache objetivo.
00058  *
00059  * @return 0 en caso de insertar un nuevo par clave-valor exitosamente, 1 si la clave ya estaba en la
00060  * cache y solo se actualizo el valor, -1 si se produjo un error.
00061  */
00061 int cache_put(void* key, size_t key_size, void* val, size_t val_size, Cache cache);
00062
00063
00064 /**
00065  * @brief Elimina el par clave-valor asociado a \a key en la cache objetivo.
00066  *
00067  * @param key La clave del par.
00068  * @param key_size El tamaño de la clave.

```



```

00069 * @param cache La cache objetivo.
00070 *
00071 * @return 0 si se elimina un par, 1 si no se encuentra el par, -1 si se
00072 * produjo un error.
00073 */
00074 int cache_delete(void* key, size_t key_size, Cache cache);
00075
00076 /**
00077 * @brief Genera un reporte de estadísticas de uso de la cache objetivo.
00078 *
00079 * @param cache La cache objetivo.
00080 * @return Un reporte con los contadores para cada operación realizada por la cache.
00081 */
00082 StatsReport cache_report(Cache cache);
00083
00084 /**
00085 * @brief Libera memoria en la cache eliminando nodos. Comienza desde los que no fueron accedidos
00086 * recientemente. Libera hasta un máximo de memoria determinado.
00087 *
00088 * @param cache La cache donde se liberará memoria.
00089 * @param memory_goal La cantidad de memoria objetivo a liberar.
00090 * @return La cantidad de memoria liberada al eliminar, que puede ser menor que el objetivo de
00091 * memoria a liberar, o -1 si se produjo un error.
00092 */
00093 ssize_t cache_free_up_memory(Cache cache, size_t memory_goal);
00094
00095 /**
00096 * @brief Obtiene el puntero a la estructura que almacena las estadísticas de la cache objetivo.
00097 *
00098 * @param cache Cache objetivo.
00099 * @return El puntero a la estructura CacheStats.
00100 */
00101 CacheStats cache_get_cstats(Cache cache);
00102
00103 /**
00104 * @brief Libera la memoria de todas las componentes de la cache, incluida
00105 * esta última.
00106 *
00107 * @param cache La cache a destruir.
00108 */
00109 void cache_destroy(Cache cache);
00110
00111 #endif // __CACHE_H__

```

4.11 cache/cache_stats.c File Reference

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "cache_stats.h"

```

Classes

- struct [CacheStats](#)

Functions

- [CacheStats cache_stats_create](#) ()
Crea un nuevo [CacheStats](#) con todos sus contadores de operaciones inicializados a 0.
- int [cache_stats_put_counter_inc](#) (CacheStats cstats)
Incrementa el contador de operaciones *PUT* del acumulador de estadísticas *cstats*.
- int [cache_stats_put_counter_dec](#) (CacheStats cstats)

- Decrementa el contador de operaciones PUT del acumulador de estadísticas cstats.*
- int [cache_stats_get_counter_inc](#) ([CacheStats](#) cstats)
- Incrementa el contador de operaciones GET del acumulador de estadísticas cstats.*
- int [cache_stats_get_counter_dec](#) ([CacheStats](#) cstats)
- Decrementa el contador de operaciones GET del acumulador de estadísticas cstats.*
- int [cache_stats_del_counter_inc](#) ([CacheStats](#) cstats)
- Incrementa el contador de operaciones DEL del acumulador de estadísticas cstats.*
- int [cache_stats_del_counter_dec](#) ([CacheStats](#) cstats)
- Decrementa el contador de operaciones DEL del acumulador de estadísticas cstats.*
- int [cache_stats_key_counter_inc](#) ([CacheStats](#) cstats)
- Incrementa el contador de claves del acumulador de estadísticas cstats.*
- int [cache_stats_key_counter_dec](#) ([CacheStats](#) cstats)
- Decrementa el contador de claves del acumulador de estadísticas cstats.*
- int [cache_stats_evict_counter_inc](#) ([CacheStats](#) cstats)
- Incrementa el contador de operaciones de expulsión (evict) del producto de la política de desalojo implementada.*
- int [cache_stats_evict_counter_dec](#) ([CacheStats](#) cstats)
- Decrementa el contador de operaciones de expulsión (evict) del producto de la política de desalojo implementada.*
- int [cache_stats_allocated_memory_add](#) ([CacheStats](#) cstats, [Counter](#) mem)
- Aumenta el acumulador de memoria asignada de la cache en mem.*
- int [cache_stats_allocated_memory_free](#) ([CacheStats](#) cstats, [Counter](#) mem)
- Subtrae rem del acumulador de memoria asignada de la cache.*
- [Counter](#) [cache_stats_get_allocated_memory](#) ([CacheStats](#) cstats)
- Obtiene la cantidad de memoria asignada dinámicamente para claves y valores de la cache asociada a cstats.abort.*
- [StatsReport](#) [cache_stats_report](#) ([CacheStats](#) cstats)
- Genera un StatsReport con información sobre las métricas de la cache al momento de ser invocada.*
- int [stats_report_stringify](#) ([StatsReport](#) report, char *buf)
- Dado un StatsReport genera un string formateado que muestra la información obtenida y lo carga en el buffer objetivo.*
- void [cache_stats_show](#) ([CacheStats](#) cstats)
- Imprime en pantalla un resumen de las estadísticas de la cache.*
- int [cache_stats_destroy](#) ([CacheStats](#) cstats)
- Destruye la estructura CacheStats apuntada por cstats, liberando la memoria que se le había asignado.*

4.11.1 Function Documentation

4.11.1.1 cache_stats_allocated_memory_add()

```
int cache_stats_allocated_memory_add (
    CacheStats cstats,
    Counter mem )
```

Aumenta el acumulador de memoria asignada de la cache en mem.

Parameters

<i>cstats</i>	Puntero a la estructura CacheStats con información de la cache.
<i>mem</i>	Cantidad de memoria a incrementar.

Returns

0 si la operacion es exitosa, -1 si se produjo un error.

Definition at line 82 of file [cache_stats.c](#).

```
00082                                     {
00083     return atom_counter_add(cstats->allocated_memory, mem);
00084 }
```

4.11.1.2 cache_stats_allocated_memory_free()

```
int cache_stats_allocated_memory_free (
    CacheStats cstats,
    Counter mem )
```

Subtrae *mem* del acumulador de memoria asignada de la cache.

Parameters

<i>cstats</i>	Puntero a la estructura CacheStats con informacion de la cache.
<i>mem</i>	Cantidad de memoria a sustraer.

Returns

0 si la operacion es exitosa, -1 si se produjo un error.

Definition at line 86 of file [cache_stats.c](#).

```
00086                                     {
00087     return atom_counter_drop(cstats->allocated_memory, mem);
00088 }
```

4.11.1.3 cache_stats_create()

```
CacheStats cache_stats_create ( )
```

Crea un nuevo [CacheStats](#) con todos sus contadores de operaciones inicializados a 0.

Returns

El nuevo struct [CacheStats](#).

Definition at line 20 of file [cache_stats.c](#).

```
00020                                     {
00021
00022     CacheStats stats = malloc(sizeof(struct CacheStats));
00023     if (stats == NULL)
00024         return NULL;
00025
00026     stats->put_counter = atom_counter_create(0);
00027     stats->get_counter = atom_counter_create(0);
00028     stats->del_counter = atom_counter_create(0);
00029
00030     stats->evict_counter = atom_counter_create(0);
00031     stats->key_counter   = atom_counter_create(0);
00032     stats->allocated_memory = atom_counter_create(0);
00033
00034     return stats;
00035
00036 }
```

4.11.1.4 `cache_stats_del_counter_dec()`

```
int cache_stats_del_counter_dec (
    CacheStats cstats )
```

Decrementa el contador de operaciones DEL del acumulador de estadísticas `cstats`.

Parameters

<code>cstats</code>	El acumulador de estadísticas objetivo.
---------------------	---

Returns

0 si la operacion es exitosa, -1 si se produjo un error.

Definition at line 60 of file `cache_stats.c`.

```
00060
00061     return atom_counter_dec(cstats->del_counter);
00062 }
```

4.11.1.5 `cache_stats_del_counter_inc()`

```
int cache_stats_del_counter_inc (
    CacheStats cstats )
```

Incrementa el contador de operaciones DEL del acumulador de estadísticas `cstats`.

Parameters

<code>cstats</code>	El acumulador de estadísticas objetivo.
---------------------	---

Returns

0 si la operacion es exitosa, -1 si se produjo un error.

Definition at line 56 of file `cache_stats.c`.

```
00056
00057     return atom_counter_inc(cstats->del_counter);
00058 }
```

4.11.1.6 `cache_stats_destroy()`

```
int cache_stats_destroy (
    CacheStats cstats )
```

Destruye la estructura `CacheStats` apuntada por `cstats`, liberando la memoria que se le habia asignado.

Parameters

<code>cstats</code>	Puntero a la estructura <code>CacheStats</code> a destruir.
---------------------	---

Returns

0 en caso de liberar exitosamente, 1 si el puntero era NULL.

Definition at line 143 of file [cache_stats.c](#).

```
00143                                     {
00144
00145     if (cstats == NULL)
00146         return 1;
00147
00148     atom_counter_destroy(cstats->put_counter);
00149     atom_counter_destroy(cstats->get_counter);
00150     atom_counter_destroy(cstats->del_counter);
00151
00152     atom_counter_destroy(cstats->evict_counter);
00153     atom_counter_destroy(cstats->key_counter);
00154
00155     free(cstats);
00156
00157     return 0;
00158
00159 }
```

4.11.1.7 cache_stats_evict_counter_dec()

```
int cache_stats_evict_counter_dec (
    CacheStats cstats )
```

Decrementa el contador de operaciones de expulsion (evict) del producto de la politica de desalojo implementada.

Parameters

<i>cstats</i>	El acumulador de estadísticas objetivo.
---------------	---

Returns

0 si la operacion es exitosa, -1 si se produjo un error.

Definition at line 78 of file [cache_stats.c](#).

```
00078                                     {
00079     return atom_counter_dec(cstats->evict_counter);
00080 }
```

4.11.1.8 cache_stats_evict_counter_inc()

```
int cache_stats_evict_counter_inc (
    CacheStats cstats )
```

Incrementa el contador de operaciones de expulsion (evict) del producto de la politica de desalojo implementada.

Parameters

<i>cstats</i>	El acumulador de estadísticas objetivo.
---------------	---

Returns

0 si la operacion es exitosa, -1 si se produjo un error.

Definition at line 74 of file [cache_stats.c](#).

```
00074 {
00075     return atom_counter_inc(cstats->evict_counter);
00076 }
```

4.11.1.9 cache_stats_get_allocated_memory()

```
Counter cache_stats_get_allocated_memory (
    CacheStats cstats )
```

Obtiene la cantidad de memoria asignada dinamicamente para claves y valores de la cache asociada a `cstats.abort`.

Parameters

<i>cstats</i>	Puntero a la estructura CacheStats con informacion de la cache.
---------------	---

Returns

La cantidad de memoria asignada dinamicamente a claves y valores.

Definition at line 90 of file [cache_stats.c](#).

```
00090 {
00091     return atom_counter_get(cstats->allocated_memory);
00092 }
```

4.11.1.10 cache_stats_get_counter_dec()

```
int cache_stats_get_counter_dec (
    CacheStats cstats )
```

Decrementa el contador de operaciones GET del acumulador de estadisticas `cstats`.

Parameters

<i>cstats</i>	El acumulador de estadisticas objetivo.
---------------	---

Returns

0 si la operacion es exitosa, -1 si se produjo un error.

Definition at line 52 of file [cache_stats.c](#).

```
00052 {
00053     return atom_counter_dec(cstats->get_counter);
00054 }
```

4.11.1.11 cache_stats_get_counter_inc()

```
int cache_stats_get_counter_inc (
    CacheStats cstats )
```

Incrementa el contador de operaciones GET del acumulador de estadísticas `cstats`.

Parameters

<code>cstats</code>	El acumulador de estadísticas objetivo.
---------------------	---

Returns

0 si la operacion es exitosa, -1 si se produjo un error.

Definition at line 48 of file `cache_stats.c`.

```
00048 {
00049     return atom_counter_inc(cstats->get_counter);
00050 }
```

4.11.1.12 cache_stats_key_counter_dec()

```
int cache_stats_key_counter_dec (
    CacheStats cstats )
```

Decrementa el contador de claves del acumulador de estadísticas `cstats`.

Parameters

<code>cstats</code>	El acumulador de estadísticas objetivo.
---------------------	---

Returns

0 si la operacion es exitosa, -1 si se produjo un error.

Definition at line 69 of file `cache_stats.c`.

```
00069 {
00070     return atom_counter_dec(cstats->key_counter);
00071 }
```

4.11.1.13 cache_stats_key_counter_inc()

```
int cache_stats_key_counter_inc (
    CacheStats cstats )
```

Incrementa el contador de claves del acumulador de estadísticas `cstats`.

Parameters

<code>cstats</code>	El acumulador de estadísticas objetivo.
---------------------	---

Returns

0 si la operacion es exitosa, -1 si se produjo un error.

Definition at line 65 of file `cache_stats.c`.

```

00065
00066     return atom_counter_inc(cstats->key_counter);
00067 }

```

4.11.1.14 cache_stats_put_counter_dec()

```

int cache_stats_put_counter_dec (
    CacheStats cstats )

```

Decrementa el contador de operaciones PUT del acumulador de estadísticas `cstats`.

Parameters

<code>cstats</code>	El acumulador de estadísticas objetivo.
---------------------	---

Returns

0 si la operacion es exitosa, -1 si se produjo un error.

Definition at line 43 of file [cache_stats.c](#).

```

00043
00044     return atom_counter_dec(cstats->put_counter);
00045 }

```

4.11.1.15 cache_stats_put_counter_inc()

```

int cache_stats_put_counter_inc (
    CacheStats cstats )

```

Incrementa el contador de operaciones PUT del acumulador de estadísticas `cstats`.

Parameters

<code>cstats</code>	El acumulador de estadísticas objetivo.
---------------------	---

Returns

0 si la operacion es exitosa, -1 si se produjo un error.

Definition at line 39 of file [cache_stats.c](#).

```

00039
00040     return atom_counter_inc(cstats->put_counter);
00041 }

```

4.11.1.16 cache_stats_report()

```

StatsReport cache_stats_report (
    CacheStats cstats )

```

Genera un [StatsReport](#) con informacion sobre las metricas de la cache al momento de ser invocada.

Parameters

<i>cstats</i>	El CacheStats asociado a la cache de interes.
---------------	---

Returns

Un [StatsReport](#) con metricas de la cache.

Definition at line 94 of file [cache_stats.c](#).

```

00094                                     {
00095
00096     StatsReport report;
00097
00098     report.put    = atom_counter_get (cstats->put_counter);
00099     report.get    = atom_counter_get (cstats->get_counter);
00100     report.del    = atom_counter_get (cstats->del_counter);
00101     report.key    = atom_counter_get (cstats->key_counter);
00102     report.evict  = atom_counter_get (cstats->evict_counter);
00103     report.allocated_memory = atom_counter_get (cstats->allocated_memory);
00104
00105     return report;
00106
00107 }
```

4.11.1.17 cache_stats_show()

```

void cache_stats_show (
    CacheStats cstats )
```

Imprime en pantalla un resumen de las estadísticas de la cache.

Parameters

<i>cstats</i>	El objeto CacheStats a resumir en pantalla
---------------	--

Definition at line 122 of file [cache_stats.c](#).

```

00122                                     {
00123
00124     printf("***** CACHE STATS *****\n");
00125     printf("PUTS: " COUNTER_FORMAT "\n", atom_counter_get (cstats->put_counter));
00126     printf("GETS: " COUNTER_FORMAT "\n", atom_counter_get (cstats->get_counter));
00127     printf("DELS: " COUNTER_FORMAT "\n", atom_counter_get (cstats->del_counter));
00128     printf("KEYS: " COUNTER_FORMAT "\n", atom_counter_get (cstats->key_counter));
00129
00130     printf("EVICTS: " COUNTER_FORMAT "\n",
00131         atom_counter_get (cstats->evict_counter));
00132
00133     printf("ALLOCATED MEMORY: " COUNTER_FORMAT "\n",
00134         atom_counter_get (cstats->allocated_memory));
00135
00136     printf("*****\n");
00137
00138     return;
00139
00140 }
```

4.11.1.18 stats_report_stringify()

```

int stats_report_stringify (
    StatsReport report,
    char * buf )
```

Dado un [StatsReport](#) genera un string formateado que muestra la informacion obtenida y lo carga en el buffer objetivo.

Parameters

<i>report</i>	El StatsReport a convertir en string.
<i>buf</i>	Puntero al buffer donde se cargara el string.

Returns

La cantidad de caracteres copiados al buffer.

Definition at line 109 of file [cache_stats.c](#).

```

00109                                     {
00110
00111     if (buf == NULL)
00112         return -1;
00113
00114     return sprintf(buf, "PUTS=%lu DELS=%lu GETS=%lu KEYS=%lu EVICTS=%lu",
00115                    report.put, report.del, report.get, report.key, report.evict);
00116
00117 }
```

4.12 cache_stats.c

[Go to the documentation of this file.](#)

```

00001 #include <stdlib.h>
00002 #include <stdio.h>
00003 #include <string.h>
00004 #include "cache_stats.h"
00005
00006 struct CacheStats {
00007
00008     AtomCounter put_counter;
00009     AtomCounter get_counter;
00010     AtomCounter del_counter;
00011
00012     AtomCounter evict_counter;
00013     AtomCounter key_counter;
00014
00015     AtomCounter allocated_memory;
00016
00017 };
00018
00019
00020 CacheStats cache_stats_create() {
00021
00022     CacheStats stats = malloc(sizeof(struct CacheStats));
00023     if (stats == NULL)
00024         return NULL;
00025
00026     stats->put_counter = atom_counter_create(0);
00027     stats->get_counter = atom_counter_create(0);
00028     stats->del_counter = atom_counter_create(0);
00029
00030     stats->evict_counter = atom_counter_create(0);
00031     stats->key_counter = atom_counter_create(0);
00032     stats->allocated_memory = atom_counter_create(0);
00033
00034     return stats;
00035
00036 }
00037
00038
00039 int cache_stats_put_counter_inc(CacheStats cstats) {
00040     return atom_counter_inc(cstats->put_counter);
00041 }
00042
00043 int cache_stats_put_counter_dec(CacheStats cstats) {
00044     return atom_counter_dec(cstats->put_counter);
00045 }
00046
00047
00048 int cache_stats_get_counter_inc(CacheStats cstats) {
00049     return atom_counter_inc(cstats->get_counter);
00050 }
```

```

00051
00052 int cache_stats_get_counter_dec(CacheStats cstats) {
00053     return atom_counter_dec(cstats->get_counter);
00054 }
00055
00056 int cache_stats_del_counter_inc(CacheStats cstats) {
00057     return atom_counter_inc(cstats->del_counter);
00058 }
00059
00060 int cache_stats_del_counter_dec(CacheStats cstats) {
00061     return atom_counter_dec(cstats->del_counter);
00062 }
00063
00064
00065 int cache_stats_key_counter_inc(CacheStats cstats) {
00066     return atom_counter_inc(cstats->key_counter);
00067 }
00068
00069 int cache_stats_key_counter_dec(CacheStats cstats) {
00070     return atom_counter_dec(cstats->key_counter);
00071 }
00072
00073
00074 int cache_stats_evict_counter_inc(CacheStats cstats) {
00075     return atom_counter_inc(cstats->evict_counter);
00076 }
00077
00078 int cache_stats_evict_counter_dec(CacheStats cstats) {
00079     return atom_counter_dec(cstats->evict_counter);
00080 }
00081
00082 int cache_stats_allocated_memory_add(CacheStats cstats, Counter mem) {
00083     return atom_counter_add(cstats->allocated_memory, mem);
00084 }
00085
00086 int cache_stats_allocated_memory_free(CacheStats cstats, Counter mem) {
00087     return atom_counter_drop(cstats->allocated_memory, mem);
00088 }
00089
00090 Counter cache_stats_get_allocated_memory(CacheStats cstats) {
00091     return atom_counter_get(cstats->allocated_memory);
00092 }
00093
00094 StatsReport cache_stats_report(CacheStats cstats) {
00095
00096     StatsReport report;
00097
00098     report.put = atom_counter_get(cstats->put_counter);
00099     report.get = atom_counter_get(cstats->get_counter);
00100     report.del = atom_counter_get(cstats->del_counter);
00101     report.key = atom_counter_get(cstats->key_counter);
00102     report.evict = atom_counter_get(cstats->evict_counter);
00103     report.allocated_memory = atom_counter_get(cstats->allocated_memory);
00104
00105     return report;
00106 }
00107
00108
00109 int stats_report_stringify(StatsReport report, char* buf) {
00110
00111     if (buf == NULL)
00112         return -1;
00113
00114     return sprintf(buf, "PUTS=%lu DELS=%lu GETS=%lu KEYS=%lu EVICTS=%lu",
00115                    report.put, report.del, report.get, report.key, report.evict);
00116 }
00117
00118
00119
00120
00121
00122 void cache_stats_show(CacheStats cstats) {
00123
00124     printf("***** CACHE STATS *****\n");
00125     printf("PUTS: " COUNTER_FORMAT "\n", atom_counter_get(cstats->put_counter));
00126     printf("GETS: " COUNTER_FORMAT "\n", atom_counter_get(cstats->get_counter));
00127     printf("DELS: " COUNTER_FORMAT "\n", atom_counter_get(cstats->del_counter));
00128     printf("KEYS: " COUNTER_FORMAT "\n", atom_counter_get(cstats->key_counter));
00129
00130     printf("EVICTS: " COUNTER_FORMAT "\n",
00131            atom_counter_get(cstats->evict_counter));
00132
00133     printf("ALLOCATED MEMORY: " COUNTER_FORMAT "\n",
00134            atom_counter_get(cstats->allocated_memory));
00135
00136     printf("*****\n");
00137

```

```

00138     return;
00139
00140 }
00141
00142
00143 int cache_stats_destroy(CacheStats cstats) {
00144
00145     if (cstats == NULL)
00146         return 1;
00147
00148     atom_counter_destroy(cstats->put_counter);
00149     atom_counter_destroy(cstats->get_counter);
00150     atom_counter_destroy(cstats->del_counter);
00151
00152     atom_counter_destroy(cstats->evict_counter);
00153     atom_counter_destroy(cstats->key_counter);
00154
00155     free(cstats);
00156
00157     return 0;
00158
00159 }

```

4.13 cache/cache_stats.h File Reference

```
#include "../helpers/atom_counter.h"
```

Classes

- struct [StatsReport](#)

Macros

- #define [STATS_COUNT](#) 6
- #define [STATS_MESSAGE_LENGTH](#) 256

Typedefs

- typedef struct [CacheStats](#) * [CacheStats](#)
- typedef struct [StatsReport](#) [StatsReport](#)

Functions

- [CacheStats](#) [cache_stats_create](#) ()
Crea un nuevo [CacheStats](#) con todos sus contadores de operaciones inicializados a 0.
- int [cache_stats_put_counter_inc](#) ([CacheStats](#) cstats)
Incrementa el contador de operaciones [PUT](#) del acumulador de estadísticas [cstats](#).
- int [cache_stats_put_counter_dec](#) ([CacheStats](#) cstats)
Decrementa el contador de operaciones [PUT](#) del acumulador de estadísticas [cstats](#).
- int [cache_stats_get_counter_inc](#) ([CacheStats](#) cstats)
Incrementa el contador de operaciones [GET](#) del acumulador de estadísticas [cstats](#).
- int [cache_stats_get_counter_dec](#) ([CacheStats](#) cstats)
Decrementa el contador de operaciones [GET](#) del acumulador de estadísticas [cstats](#).
- int [cache_stats_del_counter_inc](#) ([CacheStats](#) cstats)
Incrementa el contador de operaciones [DEL](#) del acumulador de estadísticas [cstats](#).

- int [cache_stats_del_counter_dec](#) ([CacheStats](#) cstats)
Decrementa el contador de operaciones DEL del acumulador de estadísticas cstats.
- int [cache_stats_evict_counter_inc](#) ([CacheStats](#) cstats)
Incrementa el contador de operaciones de expulsión (evict) del producto de la política de desalojo implementada.
- int [cache_stats_evict_counter_dec](#) ([CacheStats](#) cstats)
Decrementa el contador de operaciones de expulsión (evict) del producto de la política de desalojo implementada.
- int [cache_stats_key_counter_inc](#) ([CacheStats](#) cstats)
Incrementa el contador de claves del acumulador de estadísticas cstats.
- int [cache_stats_key_counter_dec](#) ([CacheStats](#) cstats)
Decrementa el contador de claves del acumulador de estadísticas cstats.
- int [cache_stats_allocated_memory_add](#) ([CacheStats](#) cstats, [Counter](#) mem)
Aumenta el acumulador de memoria asignada de la cache en mem.
- int [cache_stats_allocated_memory_free](#) ([CacheStats](#) cstats, [Counter](#) mem)
Subtrae rem del acumulador de memoria asignada de la cache.
- [Counter](#) [cache_stats_get_allocated_memory](#) ([CacheStats](#) cstats)
Obtiene la cantidad de memoria asignada dinamicamente para claves y valores de la cache asociada a cstats.abort.
- [StatsReport](#) [cache_stats_report](#) ([CacheStats](#) cstats)
Genera un StatsReport con informacion sobre las metricas de la cache al momento de ser invocada.
- int [stats_report_stringify](#) ([StatsReport](#) report, char *buf)
Dado un StatsReport genera un string formateado que muestra la informacion obtenida y lo carga en el buffer objetivo.
- void [cache_stats_show](#) ([CacheStats](#) cstats)
Imprime en pantalla un resumen de las estadísticas de la cache.
- int [cache_stats_destroy](#) ([CacheStats](#) cstats)
Destruye la estructura CacheStats apuntada por cstats, liberando la memoria que se le habia asignado.

4.13.1 Macro Definition Documentation

4.13.1.1 STATS_COUNT

```
#define STATS_COUNT 6
```

Definition at line 6 of file [cache_stats.h](#).

4.13.1.2 STATS_MESSAGE_LENGTH

```
#define STATS_MESSAGE_LENGTH 256
```

Definition at line 7 of file [cache_stats.h](#).

4.13.2 Typedef Documentation

4.13.2.1 CacheStats

```
typedef struct CacheStats* CacheStats
```

Definition at line 9 of file [cache_stats.h](#).

4.13.2.2 StatsReport

```
typedef struct StatsReport StatsReport
```

4.13.3 Function Documentation

4.13.3.1 cache_stats_allocated_memory_add()

```
int cache_stats_allocated_memory_add (
    CacheStats cstats,
    Counter mem )
```

Aumenta el acumulador de memoria asignada de la cache en `mem`.

Parameters

<i>cstats</i>	Puntero a la estructura CacheStats con informacion de la cache.
<i>mem</i>	Cantidad de memoria a incrementar.

Returns

0 si la operacion es exitosa, -1 si se produjo un error.

Definition at line 82 of file [cache_stats.c](#).

```
00082
00083     return atom_counter_add(cstats->allocated_memory, mem);
00084 }
```

4.13.3.2 cache_stats_allocated_memory_free()

```
int cache_stats_allocated_memory_free (
    CacheStats cstats,
    Counter mem )
```

Subtrae `rem` del acumulador de memoria asignada de la cache.

Parameters

<i>cstats</i>	Puntero a la estructura CacheStats con informacion de la cache.
<i>mem</i>	Cantidad de memoria a sustraer.

Returns

0 si la operacion es exitosa, -1 si se produjo un error.

Definition at line 86 of file [cache_stats.c](#).

```
00086
00087     return atom_counter_drop(cstats->allocated_memory, mem);
00088 }
```

4.13.3.3 cache_stats_create()

```
CacheStats cache_stats_create ( )
```

Crea un nuevo `CacheStats` con todos sus contadores de operaciones inicializados a 0.

Returns

El nuevo struct `CacheStats`.

Definition at line 20 of file `cache_stats.c`.

```
00020         {
00021
00022     CacheStats stats = malloc(sizeof(struct CacheStats));
00023     if (stats == NULL)
00024         return NULL;
00025
00026     stats->put_counter = atom_counter_create(0);
00027     stats->get_counter = atom_counter_create(0);
00028     stats->del_counter = atom_counter_create(0);
00029
00030     stats->evict_counter = atom_counter_create(0);
00031     stats->key_counter = atom_counter_create(0);
00032     stats->allocated_memory = atom_counter_create(0);
00033
00034     return stats;
00035
00036 }
```

4.13.3.4 cache_stats_del_counter_dec()

```
int cache_stats_del_counter_dec (
    CacheStats cstats )
```

Decrementa el contador de operaciones DEL del acumulador de estadísticas `cstats`.

Parameters

<code>cstats</code>	El acumulador de estadísticas objetivo.
---------------------	---

Returns

0 si la operacion es exitosa, -1 si se produjo un error.

Definition at line 60 of file `cache_stats.c`.

```
00060         {
00061     return atom_counter_dec(cstats->del_counter);
00062 }
```

4.13.3.5 cache_stats_del_counter_inc()

```
int cache_stats_del_counter_inc (
    CacheStats cstats )
```

Incrementa el contador de operaciones DEL del acumulador de estadísticas `cstats`.

Parameters

<code>cstats</code>	El acumulador de estadísticas objetivo.
---------------------	---

Returns

0 si la operacion es exitosa, -1 si se produjo un error.

Definition at line 56 of file [cache_stats.c](#).

```
00056 {
00057     return atom_counter_inc(cstats->del_counter);
00058 }
```

4.13.3.6 cache_stats_destroy()

```
int cache_stats_destroy (
    CacheStats cstats )
```

Destruye la estructura [CacheStats](#) apuntada por `cstats`, liberando la memoria que se le habia asignado.

Parameters

<code>cstats</code>	Puntero a la estructura CacheStats a destruir.
---------------------	--

Returns

0 en caso de liberar exitosamente, 1 si el puntero era NULL.

Definition at line 143 of file [cache_stats.c](#).

```
00143 {
00144
00145     if (cstats == NULL)
00146         return 1;
00147
00148     atom_counter_destroy(cstats->put_counter);
00149     atom_counter_destroy(cstats->get_counter);
00150     atom_counter_destroy(cstats->del_counter);
00151
00152     atom_counter_destroy(cstats->evict_counter);
00153     atom_counter_destroy(cstats->key_counter);
00154
00155     free(cstats);
00156
00157     return 0;
00158
00159 }
```

4.13.3.7 cache_stats_evict_counter_dec()

```
int cache_stats_evict_counter_dec (
    CacheStats cstats )
```

Decrementa el contador de operaciones de expulsion (`evict`) del producto de la politica de desalojo implementada.

Parameters

<code>cstats</code>	El acumulador de estadísticas objetivo.
---------------------	---

Returns

0 si la operación es exitosa, -1 si se produjo un error.

Definition at line 78 of file [cache_stats.c](#).

```
00078 {
00079     return atom_counter_dec(cstats->evict_counter);
00080 }
```

4.13.3.8 cache_stats_evict_counter_inc()

```
int cache_stats_evict_counter_inc (
    CacheStats cstats )
```

Incrementa el contador de operaciones de expulsión (`evict`) del producto de la política de desalojo implementada.

Parameters

<code>cstats</code>	El acumulador de estadísticas objetivo.
---------------------	---

Returns

0 si la operación es exitosa, -1 si se produjo un error.

Definition at line 74 of file [cache_stats.c](#).

```
00074 {
00075     return atom_counter_inc(cstats->evict_counter);
00076 }
```

4.13.3.9 cache_stats_get_allocated_memory()

```
Counter cache_stats_get_allocated_memory (
    CacheStats cstats )
```

Obtiene la cantidad de memoria asignada dinámicamente para claves y valores de la cache asociada a `cstats.abort`.

Parameters

<code>cstats</code>	Puntero a la estructura CacheStats con información de la cache.
---------------------	---

Returns

La cantidad de memoria asignada dinámicamente a claves y valores.

Definition at line 90 of file [cache_stats.c](#).

```
00090 {
00091     return atom_counter_get(cstats->allocated_memory);
00092 }
```

4.13.3.10 `cache_stats_get_counter_dec()`

```
int cache_stats_get_counter_dec (  
    CacheStats cstats )
```

Decrementa el contador de operaciones GET del acumulador de estadísticas `cstats`.

Parameters

<code>cstats</code>	El acumulador de estadísticas objetivo.
---------------------	---

Returns

0 si la operacion es exitosa, -1 si se produjo un error.

Definition at line 52 of file `cache_stats.c`.

```
00052  
00053     return atom_counter_dec(cstats->get_counter);  
00054 }
```

4.13.3.11 `cache_stats_get_counter_inc()`

```
int cache_stats_get_counter_inc (  
    CacheStats cstats )
```

Incrementa el contador de operaciones GET del acumulador de estadísticas `cstats`.

Parameters

<code>cstats</code>	El acumulador de estadísticas objetivo.
---------------------	---

Returns

0 si la operacion es exitosa, -1 si se produjo un error.

Definition at line 48 of file `cache_stats.c`.

```
00048  
00049     return atom_counter_inc(cstats->get_counter);  
00050 }
```

4.13.3.12 `cache_stats_key_counter_dec()`

```
int cache_stats_key_counter_dec (  
    CacheStats cstats )
```

Decrementa el contador de claves del acumulador de estadísticas `cstats`.

Parameters

<code>cstats</code>	El acumulador de estadísticas objetivo.
---------------------	---

Returns

0 si la operacion es exitosa, -1 si se produjo un error.

Definition at line 69 of file [cache_stats.c](#).

```
00069 {  
00070     return atom_counter_dec(cstats->key_counter);  
00071 }
```

4.13.3.13 cache_stats_key_counter_inc()

```
int cache_stats_key_counter_inc (  
    CacheStats cstats )
```

Incrementa el contador de claves del acumulador de estadísticas `cstats`.

Parameters

<code>cstats</code>	El acumulador de estadísticas objetivo.
---------------------	---

Returns

0 si la operacion es exitosa, -1 si se produjo un error.

Definition at line 65 of file [cache_stats.c](#).

```
00065 {  
00066     return atom_counter_inc(cstats->key_counter);  
00067 }
```

4.13.3.14 cache_stats_put_counter_dec()

```
int cache_stats_put_counter_dec (  
    CacheStats cstats )
```

Decrementa el contador de operaciones PUT del acumulador de estadísticas `cstats`.

Parameters

<code>cstats</code>	El acumulador de estadísticas objetivo.
---------------------	---

Returns

0 si la operacion es exitosa, -1 si se produjo un error.

Definition at line 43 of file [cache_stats.c](#).

```
00043 {  
00044     return atom_counter_dec(cstats->put_counter);  
00045 }
```

4.13.3.15 cache_stats_put_counter_inc()

```
int cache_stats_put_counter_inc (  
    CacheStats cstats )
```

Incrementa el contador de operaciones `PUT` del acumulador de estadísticas `cstats`.

Parameters

<code>cstats</code>	El acumulador de estadísticas objetivo.
---------------------	---

Returns

0 si la operación es exitosa, -1 si se produjo un error.

Definition at line 39 of file `cache_stats.c`.

```
00039 {
00040     return atom_counter_inc(cstats->put_counter);
00041 }
```

4.13.3.16 cache_stats_report()

```
StatsReport cache_stats_report (
    CacheStats cstats )
```

Genera un `StatsReport` con información sobre las métricas de la cache al momento de ser invocada.

Parameters

<code>cstats</code>	El <code>CacheStats</code> asociado a la cache de interés.
---------------------	--

Returns

Un `StatsReport` con métricas de la cache.

Definition at line 94 of file `cache_stats.c`.

```
00094 {
00095
00096     StatsReport report;
00097
00098     report.put    = atom_counter_get(cstats->put_counter);
00099     report.get    = atom_counter_get(cstats->get_counter);
00100     report.del    = atom_counter_get(cstats->del_counter);
00101     report.key    = atom_counter_get(cstats->key_counter);
00102     report.evict  = atom_counter_get(cstats->evict_counter);
00103     report.allocated_memory = atom_counter_get(cstats->allocated_memory);
00104
00105     return report;
00106
00107 }
```

4.13.3.17 cache_stats_show()

```
void cache_stats_show (
    CacheStats cstats )
```

Imprime en pantalla un resumen de las estadísticas de la cache.

Parameters

<code>cstats</code>	El objeto <code>CacheStats</code> a resumir en pantalla
---------------------	---

Definition at line 122 of file [cache_stats.c](#).

```
00122                                     {
00123
00124     printf("***** CACHE STATS *****\n");
00125     printf("PUTS: " COUNTER_FORMAT "\n", atom_counter_get(cstats->put_counter));
00126     printf("GETS: " COUNTER_FORMAT "\n", atom_counter_get(cstats->get_counter));
00127     printf("DELS: " COUNTER_FORMAT "\n", atom_counter_get(cstats->del_counter));
00128     printf("KEYS: " COUNTER_FORMAT "\n", atom_counter_get(cstats->key_counter));
00129
00130     printf("EVICTS: " COUNTER_FORMAT "\n",
00131           atom_counter_get(cstats->evict_counter));
00132
00133     printf("ALLOCATED MEMORY: " COUNTER_FORMAT "\n",
00134           atom_counter_get(cstats->allocated_memory));
00135
00136     printf("*****\n");
00137
00138     return;
00139
00140 }
```

4.13.3.18 stats_report_stringify()

```
int stats_report_stringify (
    StatsReport report,
    char * buf )
```

Dado un [StatsReport](#) genera un string formateado que muestra la informacion obtenida y lo carga en el buffer objetivo.

Parameters

<i>report</i>	El StatsReport a convertir en string.
<i>buf</i>	Puntero al buffer donde se cargara el string.

Returns

La cantidad de caracteres copiados al buffer.

Definition at line 109 of file [cache_stats.c](#).

```
00109                                     {
00110
00111     if (buf == NULL)
00112         return -1;
00113
00114     return sprintf(buf, "PUTS=%lu DELS=%lu GETS=%lu KEYS=%lu EVICTS=%lu",
00115                   report.put, report.del, report.get, report.key, report.evict);
00116
00117 }
```

4.14 cache_stats.h

[Go to the documentation of this file.](#)

```
00001 #ifndef __CACHE_STATS_H__
00002 #define __CACHE_STATS_H__
00003
00004 #include "../helpers/atom_counter.h"
00005
00006 #define STATS_COUNT 6 // Es la cantidad de estadisticos que reportamos.
00007 #define STATS_MESSAGE_LENGTH 256
00008
00009 typedef struct CacheStats* CacheStats;
00010
00011 typedef struct StatsReport {
00012     Counter put;
```

```

00013     Counter get;
00014     Counter del;
00015     Counter key;
00016     Counter evict;
00017     Counter allocated_memory;
00018 } StatsReport;
00019
00020 /**
00021  * @brief Crea un nuevo `CacheStats` con todos sus contadores de operaciones inicializados a 0.
00022  * @return El nuevo struct CacheStats.
00023  */
00024 CacheStats cache_stats_create();
00025
00026 /**
00027  * @brief Incrementa el contador de operaciones `PUT` del acumulador de estadísticas `cstats`.
00028  * @param cstats El acumulador de estadísticas objetivo.
00029  * @return 0 si la operación es exitosa, -1 si se produjo un error.
00030  */
00031 int cache_stats_put_counter_inc(CacheStats cstats);
00032
00033 /**
00034  * @brief Decrementa el contador de operaciones `PUT` del acumulador de estadísticas `cstats`.
00035  * @param cstats El acumulador de estadísticas objetivo.
00036  * @return 0 si la operación es exitosa, -1 si se produjo un error.
00037  */
00038 int cache_stats_put_counter_dec(CacheStats cstats);
00039
00040 /**
00041  * @brief Incrementa el contador de operaciones `GET` del acumulador de estadísticas `cstats`.
00042  * @param cstats El acumulador de estadísticas objetivo.
00043  * @return 0 si la operación es exitosa, -1 si se produjo un error.
00044  */
00045 int cache_stats_get_counter_inc(CacheStats cstats);
00046
00047 /**
00048  * @brief Decrementa el contador de operaciones `GET` del acumulador de estadísticas `cstats`.
00049  * @param cstats El acumulador de estadísticas objetivo.
00050  * @return 0 si la operación es exitosa, -1 si se produjo un error.
00051  */
00052 int cache_stats_get_counter_dec(CacheStats cstats);
00053
00054 /**
00055  * @brief Incrementa el contador de operaciones `DEL` del acumulador de estadísticas `cstats`.
00056  * @param cstats El acumulador de estadísticas objetivo.
00057  * @return 0 si la operación es exitosa, -1 si se produjo un error.
00058  */
00059 int cache_stats_del_counter_inc(CacheStats cstats);
00060
00061 /**
00062  * @brief Decrementa el contador de operaciones `DEL` del acumulador de estadísticas `cstats`.
00063  * @param cstats El acumulador de estadísticas objetivo.
00064  * @return 0 si la operación es exitosa, -1 si se produjo un error.
00065  */
00066 int cache_stats_del_counter_dec(CacheStats cstats);
00067
00068 /**
00069  * @brief Incrementa el contador de operaciones de expulsión (`evict`) del producto de la política
00070  * de desalojo implementada.
00071  * @param cstats El acumulador de estadísticas objetivo.
00072  * @return 0 si la operación es exitosa, -1 si se produjo un error.
00073  */
00074 int cache_stats_evict_counter_inc(CacheStats cstats);
00075
00076 /**
00077  * @brief Decrementa el contador de operaciones de expulsión (`evict`) del producto de la política
00078  * de desalojo implementada.
00079  * @param cstats El acumulador de estadísticas objetivo.
00080  * @return 0 si la operación es exitosa, -1 si se produjo un error.
00081  */
00082 int cache_stats_evict_counter_dec(CacheStats cstats);
00083
00084 /**
00085  * @brief Incrementa el contador de claves del acumulador de estadísticas `cstats`.
00086  * @param cstats El acumulador de estadísticas objetivo.
00087  * @return 0 si la operación es exitosa, -1 si se produjo un error.
00088  */
00089 int cache_stats_key_counter_inc(CacheStats cstats);
00090
00091

```

```

00098 /**
00099  * @brief Decrementa el contador de claves del acumulador de estadísticas `cstats`.
00100  *
00101  * @param cstats El acumulador de estadísticas objetivo.
00102  * @return 0 si la operación es exitosa, -1 si se produjo un error.
00103  */
00104 int cache_stats_key_counter_dec(CacheStats cstats);
00105
00106 /**
00107  * @brief Aumenta el acumulador de memoria asignada de la cache en `mem`.
00108  *
00109  * @param cstats Puntero a la estructura CacheStats con información de la cache.
00110  * @param mem Cantidad de memoria a incrementar.
00111  *
00112  * @return 0 si la operación es exitosa, -1 si se produjo un error.
00113  */
00114 int cache_stats_allocated_memory_add(CacheStats cstats, Counter mem);
00115
00116
00117 /**
00118  * @brief Subtrae `rem` del acumulador de memoria asignada de la cache.
00119  *
00120  * @param cstats Puntero a la estructura CacheStats con información de la cache.
00121  * @param mem Cantidad de memoria a sustraer.
00122  *
00123  * @return 0 si la operación es exitosa, -1 si se produjo un error.
00124  */
00125 int cache_stats_allocated_memory_free(CacheStats cstats, Counter mem);
00126
00127 /**
00128  * @brief Obtiene la cantidad de memoria asignada dinámicamente para claves y valores de la cache
00129  * asociada a `cstats`.abort
00130  *
00131  * @param cstats Puntero a la estructura CacheStats con información de la cache.
00132  * @return La cantidad de memoria asignada dinámicamente a claves y valores.
00133  */
00133 Counter cache_stats_get_allocated_memory(CacheStats cstats);
00134
00135
00136
00137 /**
00138  * @brief Genera un StatsReport con información sobre las métricas de la cache al momento de ser
00139  * invocada.
00140  *
00141  * @param cstats El CacheStats asociado a la cache de interés.
00142  * @return Un StatsReport con métricas de la cache.
00143  */
00143 StatsReport cache_stats_report(CacheStats cstats);
00144
00145
00146 /**
00146  * @brief Dado un StatsReport genera un string formateado que muestra la información obtenida y lo
00147  * carga en el buffer objetivo.
00148  *
00149  * @param report El StatsReport a convertir en string.
00150  * @param buf Puntero al buffer donde se cargará el string.
00151  *
00152  * @return La cantidad de caracteres copiados al buffer.
00153  */
00153 int stats_report_stringify(StatsReport report, char* buf);
00154
00155
00156 /**
00156  * @brief Imprime en pantalla un resumen de las estadísticas de la cache.
00157  *
00158  * @param cstats El objeto CacheStats a resumir en pantalla
00159  *
00160  */
00160 void cache_stats_show(CacheStats cstats);
00161
00162
00163
00164 /**
00165  * @brief Destruye la estructura CacheStats apuntada por `cstats`, liberando la memoria que se le
00166  * había asignado.
00167  *
00168  * @param cstats Puntero a la estructura CacheStats a destruir.
00169  * @return 0 en caso de liberar exitosamente, 1 si el puntero era NULL.
00170  */
00170 int cache_stats_destroy(CacheStats cstats);
00171
00172
00173
00174 #endif // __CACHE_STATS_H__

```


4.15 dynalloc/dynalloc.c File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "dynalloc.h"
#include "../cache/cache.h"
```

Macros

- `#define DYNALLOC_FAIL_RATE 0`
- `#define MAX_ATTEMPTS 2`
- `#define SIZE_GOAL_FACTOR 2`

Functions

- `void * dynalloc (size_t sz, Cache cache)`

Asigna un bloque de memoria. De no contarse con memoria disponible, elimina los nodos menos recientemente utilizados de la cache hasta contar con espacio suficiente.

4.15.1 Macro Definition Documentation

4.15.1.1 DYNALLOC_FAIL_RATE

```
#define DYNALLOC_FAIL_RATE 0
```

Definition at line 7 of file [dynalloc.c](#).

4.15.1.2 MAX_ATTEMPTS

```
#define MAX_ATTEMPTS 2
```

Definition at line 8 of file [dynalloc.c](#).

4.15.1.3 SIZE_GOAL_FACTOR

```
#define SIZE_GOAL_FACTOR 2
```

Definition at line 9 of file [dynalloc.c](#).

4.15.2 Function Documentation

4.15.2.1 dynalloc()

```
void * dynalloc (
    size_t sz,
    Cache cache )
```

Asigna un bloque de memoria. De no contarse con memoria disponible, elimina los nodos menos recientemente utilizados de la cache hasta contar con espacio suficiente.

Parameters

<i>sz</i>	El tamaño del bloque a asignar.
<i>cache</i>	La cache asociada a quien invoca a esta funcion.

Returns

Un puntero a un bloque de memoria de tamaño *sz*, o NULL si se produjo un error al desalojar o alojar memoria para satisfacer el pedido.

Definition at line 13 of file [dynalloc.c](#).

```

00013                                     {
00014
00015     if (DYNALLOC_FAIL_RATE > 0 && (rand() % 100) < DYNALLOC_FAIL_RATE)
00016         PRINT("Falla de dynalloc simulada.");
00017     else {
00018         void* ptr = malloc(sz);
00019         if (ptr != NULL)
00020             return ptr;
00021     }
00022
00023     PRINT("No hay memoria suficiente. Debemos liberar memoria.");
00024
00025     // Liberaremos el maximo entre el 20% de la memoria ocupada actual y el size del bloque a asignar
por un size_factor
00026     size_t memory_goal = max(cache_stats_get_allocated_memory(
00027                             cache_get_cstats(cache)) / 5,
00028                             sz * SIZE_GOAL_FACTOR);
00029     size_t total_freed_memory = 0;
00030     ssize_t freed_memory;
00031
00032     PRINT("Allocated memory: %lu", cache_stats_get_allocated_memory(cache_get_cstats(cache)));
00033     PRINT("Memory goal: %lu", memory_goal);
00034
00035     int attempts = 0;
00036
00037     while (total_freed_memory < memory_goal && attempts < MAX_ATTEMPTS) {
00038
00039         freed_memory = cache_free_up_memory(cache, memory_goal - total_freed_memory);
00040
00041         // Si se produjo algun error, directamente retornamos NULL
00042         if (freed_memory < 0)
00043             return NULL;
00044
00045         if (freed_memory == 0) {
00046             sched_yield();
00047             attempts++;
00048         }
00049
00050         total_freed_memory += freed_memory;
00051     }
00052
00053     PRINT("Memoria liberada correctamente. Total liberado: %lu", total_freed_memory);
00054
00055     // Tanto si se logro el objetivo de memoria como si se agotaron los intentos, devolvemos
directamente malloc.
00056     // Si no alcanza la memoria, aceptamos que devuelva NULL pues la cache debe estar sobrecargada de
pedidos.
00057     return malloc(sz);
00058
00059 }
```

4.16 dynalloc.c

[Go to the documentation of this file.](#)

```

00001 #include <stdio.h>
00002 #include <stdlib.h>
00003 #include <pthread.h>
00004 #include "dynalloc.h"
00005 #include "../cache/cache.h"
00006
00007 #define DYNALLOC_FAIL_RATE 0
00008 #define MAX_ATTEMPTS 2
```

```

00009 #define SIZE_GOAL_FACTOR 2
00010
00011 static size_t max(size_t goal, size_t sz) { return sz < goal ? goal : sz; }
00012
00013 void* dynalloc(size_t sz, Cache cache) {
00014     if (DYNALLOC_FAIL_RATE > 0 && (rand() % 100) < DYNALLOC_FAIL_RATE)
00015         PRINT("Falla de dynalloc simulada.");
00016     else {
00017         void* ptr = malloc(sz);
00018         if (ptr != NULL)
00019             return ptr;
00020     }
00021
00022     PRINT("No hay memoria suficiente. Debemos liberar memoria.");
00023
00024     // Liberaremos el maximo entre el 20% de la memoria ocupada actual y el size del bloque a asignar
00025     por un size_factor
00026     size_t memory_goal = max(cache_stats_get_allocated_memory(
00027         cache_get_cstats(cache)) / 5,
00028         sz * SIZE_GOAL_FACTOR);
00029     size_t total_freed_memory = 0;
00030     ssize_t freed_memory;
00031
00032     PRINT("Allocated memory: %lu", cache_stats_get_allocated_memory(cache_get_cstats(cache)));
00033     PRINT("Memory goal: %lu", memory_goal);
00034
00035     int attempts = 0;
00036
00037     while (total_freed_memory < memory_goal && attempts < MAX_ATTEMPTS) {
00038         freed_memory = cache_free_up_memory(cache, memory_goal - total_freed_memory);
00039
00040         // Si se produjo algun error, directamente retornamos NULL
00041         if (freed_memory < 0)
00042             return NULL;
00043
00044         if (freed_memory == 0) {
00045             sched_yield();
00046             attempts++;
00047         }
00048
00049         total_freed_memory += freed_memory;
00050     }
00051
00052     PRINT("Memoria liberada correctamente. Total liberado: %lu", total_freed_memory);
00053
00054     // Tanto si se logro el objetivo de memoria como si se agotaron los intentos, devolvemos
00055     directamente malloc.
00056     // Si no alcanza la memoria, aceptamos que devuelva NULL pues la cache debe estar sobrecargada de
00057     pedidos.
00058     return malloc(sz);
00059 }

```

4.17 dynalloc/dynalloc.h File Reference

```

#include <stdlib.h>
#include <pthread.h>

```

Typedefs

- typedef struct [Cache](#) * [Cache](#)

Functions

- void * [dynalloc](#) (size_t sz, [Cache](#) cache)

Asigna un bloque de memoria. De no contarse con memoria disponible, elimina los nodos menos recientemente utilizados de la cache hasta contar con espacio suficiente.

4.17.1 Typedef Documentation

4.17.1.1 Cache

```
typedef struct Cache* Cache
```

Definition at line 8 of file [dynalloc.h](#).

4.17.2 Function Documentation

4.17.2.1 dynalloc()

```
void * dynalloc (
    size_t sz,
    Cache cache )
```

Asigna un bloque de memoria. De no contarse con memoria disponible, elimina los nodos menos recientemente utilizados de la cache hasta contar con espacio suficiente.

Parameters

<i>sz</i>	El tamaño del bloque a asignar.
<i>cache</i>	La cache asociada a quien invoca a esta funcion.

Returns

Un puntero a un bloque de memoria de tamaño *sz*, o NULL si se produjo un error al desalojar o alojar memoria para satisfacer el pedido.

Definition at line 13 of file [dynalloc.c](#).

```
00013                                     {
00014
00015     if (DYNALLOC_FAIL_RATE > 0 && (rand() % 100) < DYNALLOC_FAIL_RATE)
00016         PRINT("Falla de dynalloc simulada.");
00017     else {
00018         void* ptr = malloc(sz);
00019         if (ptr != NULL)
00020             return ptr;
00021     }
00022
00023     PRINT("No hay memoria suficiente. Debemos liberar memoria.");
00024
00025     // Liberaremos el maximo entre el 20% de la memoria ocupada actual y el size del bloque a asignar
    por un size_factor
00026     size_t memory_goal = max(cache_stats_get_allocated_memory(
00027                             cache_get_cstats(cache)) / 5,
00028                             sz * SIZE_GOAL_FACTOR);
00029     size_t total_freed_memory = 0;
00030     ssize_t freed_memory;
00031
00032     PRINT("Allocated memory: %lu", cache_stats_get_allocated_memory(cache_get_cstats(cache)));
00033     PRINT("Memory goal: %lu", memory_goal);
00034
00035     int attempts = 0;
00036
00037     while (total_freed_memory < memory_goal && attempts < MAX_ATTEMPTS) {
00038
00039         freed_memory = cache_free_up_memory(cache, memory_goal - total_freed_memory);
00040
00041         // Si se produjo algun error, directamente retornamos NULL
00042         if (freed_memory < 0)
00043             return NULL;
00044         total_freed_memory += freed_memory;
00045         attempts++;
00046     }
```

```

00045         if (freed_memory == 0) {
00046             sched_yield();
00047             attempts++;
00048         }
00049
00050         total_freed_memory += freed_memory;
00051     }
00052
00053     PRINT("Memoria liberada correctamente. Total liberado: %lu", total_freed_memory);
00054
00055     // Tanto si se logro el objetivo de memoria como si se agotaron los intentos, devolvemos
    directamente malloc.
00056     // Si no alcanza la memoria, aceptamos que devuelva NULL pues la cache debe estar sobrecargada de
    pedidos.
00057     return malloc(sz);
00058
00059 }

```

4.18 dynalloc.h

[Go to the documentation of this file.](#)

```

00001 #ifndef __DYNALLOC_H__
00002 #define __DYNALLOC_H__
00003
00004 #include <stdlib.h>
00005 #include <pthread.h>
00006
00007 // forward declaration para no hacer el include cache.h
00008 typedef struct Cache* Cache;
00009
00010
00011 /**
00012  * @brief Asigna un bloque de memoria. De no contarse con memoria disponible, elimina los nodos menos
    recientemente utilizados de la cache hasta contar con espacio suficiente.
00013  *
00014  * @param sz El tamaño del bloque a asignar.
00015  * @param cache La cache asociada a quien invoca a esta funcion.
00016  *
00017  * @return Un puntero a un bloque de memoria de tamaño `sz`, o NULL si se produjo un error al
    desalojar o alojar memoria para satisfacer el pedido.
00018  */
00019 void* dynalloc(size_t sz, Cache cache);
00020
00021
00022 #endif // __DYNALLOC_H__

```

4.19 hashmap/hash.c File Reference

```
#include "hash.h"
```

Functions

- unsigned long [kr_hash](#) (char *key, size_t size)
- unsigned long [dek_hash](#) (char *key, size_t size)

4.19.1 Function Documentation

4.19.1.1 dek_hash()

```

unsigned long dek_hash (
    char * key,
    size_t size )

```

Definition at line 14 of file [hash.c](#).

```
00015 {
00016     unsigned long hash = size;
00017     while (*key)
00018     {
00019         hash <<= 5;
00020         hash ^= (hash >> 27);
00021         hash ^= *key;
00022         key++;
00023     }
00024     return hash;
00025 }
```

4.19.1.2 kr_hash()

```
unsigned long kr_hash (
    char * key,
    size_t size )
```

Definition at line 3 of file [hash.c](#).

```
00003                                     {
00004
00005     unsigned long hashval;
00006     unsigned long i;
00007
00008     for (i = 0, hashval = 0 ; i < size ; ++i, key++)
00009         hashval = *key + 31 * hashval;
00010
00011     return hashval;
00012 }
```

4.20 hash.c

[Go to the documentation of this file.](#)

```
00001 #include "hash.h"
00002
00003 unsigned long kr_hash(char* key, size_t size) {
00004
00005     unsigned long hashval;
00006     unsigned long i;
00007
00008     for (i = 0, hashval = 0 ; i < size ; ++i, key++)
00009         hashval = *key + 31 * hashval;
00010
00011     return hashval;
00012 }
00013
00014 unsigned long dek_hash(char *key, size_t size)
00015 {
00016     unsigned long hash = size;
00017     while (*key)
00018     {
00019         hash <<= 5;
00020         hash ^= (hash >> 27);
00021         hash ^= *key;
00022         key++;
00023     }
00024     return hash;
00025 }
```

4.21 hashmap/hash.h File Reference

```
#include <stdlib.h>
```

Functions

- unsigned long [kr_hash](#) (char *key, size_t size)
- unsigned long [dek_hash](#) (char *key, size_t size)

4.21.1 Function Documentation

4.21.1.1 dek_hash()

```
unsigned long dek_hash (
    char * key,
    size_t size )
```

Definition at line 14 of file [hash.c](#).

```
00015 {
00016     unsigned long hash = size;
00017     while (*key)
00018     {
00019         hash «= 5;
00020         hash ^= (hash » 27);
00021         hash ^= *key;
00022         key++;
00023     }
00024     return hash;
00025 }
```

4.21.1.2 kr_hash()

```
unsigned long kr_hash (
    char * key,
    size_t size )
```

Definition at line 3 of file [hash.c](#).

```
00003                                     {
00004
00005     unsigned long hashval;
00006     unsigned long i;
00007
00008     for (i = 0, hashval = 0 ; i < size ; ++i, key++)
00009         hashval = *key + 31 * hashval;
00010
00011     return hashval;
00012 }
```

4.22 hash.h

[Go to the documentation of this file.](#)

```
00001 #ifndef __HASH_H__
00002 #define __HASH_H__
00003
00004 #include <stdlib.h>
00005
00006 /* Funcion de hash de K&R */
00007
00008 unsigned long kr_hash(char* key, size_t size);
00009
00010 unsigned long dek_hash(char *key, size_t size);
00011
00012 #endif // __HASH_H__
```

4.23 hashmap/hashnode.c File Reference

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>
#include "hashnode.h"
#include "../lru/lrunode.h"
```

Classes

- struct [HashNode](#)

Functions

- int [hashnode_keys_equal](#) (const void *key_a, size_t size_a, const void *key_b, size_t size_b)
- [HashNode hashnode_create](#) (void *key, size_t key_size, void *val, size_t val_size, [Cache](#) cache)
Crea un nuevo nodo.
- void [hashnode_destroy](#) ([HashNode](#) node)
Destruye el nodo objetivo, liberando la memoria asociada. Se asume que el nodo ya esta 'limpio'.
- [LookupResult hashnode_lookup](#) (void *key, size_t size, [HashNode](#) node)
Busca el valor asociado a la clave en el bucket iniciado en node.
- [HashNode hashnode_lookup_node](#) (void *key, size_t size, [HashNode](#) node)
Busca el puntero al nodo asociado a la clave en el bucket iniciado en node.
- int [hashnode_clean](#) ([HashNode](#) node)
Desconecta al nodo objetivo de sus vecinos, reconectando a sus nodos adyacentes.
- void * [hashnode_get_key](#) ([HashNode](#) node)
Obtiene la clave del nodo.
- int [hashnode_set_key](#) ([HashNode](#) node, void *key, size_t new_key_size)
Establece la clave del nodo, liberando la memoria de la clave anterior, en caso de existir.
- void * [hashnode_get_val](#) ([HashNode](#) node)
Obtiene el valor del nodo.
- int [hashnode_set_val](#) ([HashNode](#) node, void *val, size_t new_val_size)
Establece la clave del nodo, liberando la memoria del valor anterior, en caso de existir.
- size_t [hashnode_get_key_size](#) ([HashNode](#) node)
Obtiene el tamaño de la clave del nodo.
- void [hashnode_set_key_size](#) ([HashNode](#) node, size_t key_size)
Establece el tamaño de la clave del nodo.
- size_t [hashnode_get_val_size](#) ([HashNode](#) node)
Obtiene el tamaño del valor del nodo.
- void [hashnode_set_val_size](#) ([HashNode](#) node, size_t val_size)
Establece el tamaño del valor del nodo.
- [HashNode hashnode_get_prev](#) ([HashNode](#) node)
Obtiene el nodo previo.
- void [hashnode_set_prev](#) ([HashNode](#) node, [HashNode](#) prev)
Establece el nodo previo.
- [HashNode hashnode_get_next](#) ([HashNode](#) node)
Obtiene el nodo siguiente.
- void [hashnode_set_next](#) ([HashNode](#) node, [HashNode](#) next)
Establece el nodo siguiente.
- [LRUNode hashnode_get_prio](#) ([HashNode](#) node)
Obtiene el puntero al nodo de prioridad asociado.
- void [hashnode_set_prio](#) ([HashNode](#) node, [LRUNode](#) prio)
Establece el puntero al nodo de prioridad.

4.23.1 Function Documentation

4.23.1.1 hashnode_clean()

```
int hashnode_clean (
    HashNode node )
```

Desconecta al nodo objetivo de sus vecinos, reconectando a sus nodos adyacentes.

IMPORTANTE: Esta funcion NO es thread-safe. Debe pedirse el mutex asociado a la clave antes de invocarla, y liberarlo tras terminada la ejecucion.

Parameters

<i>node</i>	Nodo a limpiar.
-------------	-----------------

Returns

0 si es exitoso, 1 si se produjo un error.

Definition at line 95 of file [hashnode.c](#).

```
00095                                     {
00096
00097     if (node == NULL)
00098         return -1;
00099
00100     // Desconectamos y reconectamos los adyacentes
00101     HashNode prev = hashnode_get_prev(node);
00102     HashNode next = hashnode_get_next(node);
00103
00104     hashnode_set_next(prev, next);
00105     hashnode_set_prev(next, prev);
00106
00107     return 0;
00108
00109 }
```

4.23.1.2 hashnode_create()

```
HashNode hashnode_create (
    void * key,
    size_t key_size,
    void * val,
    size_t val_size,
    Cache cache )
```

Crea un nuevo nodo.

Parameters

<i>key</i>	El puntero a la clave del nodo.
<i>key_size</i>	El tamaño de la clave.
<i>val</i>	El puntero al valor del nodo.
<i>val_size</i>	El tamaño del valor.
<i>cache</i>	Puntero a la cache donde se insertara el nodo.

Returns

Un puntero al nodo creado, que es NULL si falla al asignarse memoria.

Definition at line 22 of file [hashnode.c](#).

```

00022                                     {
00023
00024     // No es posible insertar un par sin una key que lo identifique.
00025     if (key == NULL) return NULL;
00026
00027     HashNode node = dynalloc(sizeof(struct HashNode), cache);
00028     if (node == NULL)
00029         return NULL;
00030
00031     memset(node, 0, sizeof(struct HashNode));
00032
00033     LRUNode prio = lrnode_create(cache);
00034     if (prio == NULL) {
00035         free(node);
00036         return NULL;
00037     }
00038
00039     // Setteamos la clave
00040     node->key = key;
00041     node->key_size = key_size;
00042
00043     // Asignamos memoria y setteamos el valor
00044     node->val = val;
00045     node->val_size = val_size;
00046
00047     // Setteamos la prioridad
00048     node->prio = prio;
00049
00050     return node;
00051
00052 }
```

4.23.1.3 hashnode_destroy()

```

void hashnode_destroy (
    HashNode node )
```

Destruye el nodo objetivo, liberando la memoria asociada. Se asume que el nodo ya esta 'limpio'.

Parameters

<i>node</i>	El nodo a destruir.
-------------	---------------------

Definition at line 54 of file [hashnode.c](#).

```

00054                                     {
00055
00056     if (node == NULL)
00057         return;
00058
00059     // Liberamos la clave y el valor.
00060     if (node->key)
00061         free(node->key);
00062
00063     if (node->val)
00064         free(node->val);
00065
00066     free(node);
00067
00068 }
```

4.23.1.4 hashnode_get_key()

```

void * hashnode_get_key (
    HashNode node )
```

Obtiene la clave del nodo.

Parameters

<i>node</i>	Puntero al nodo.
-------------	------------------

Returns

El puntero a la clave del nodo.

Definition at line 124 of file [hashnode.c](#).

```
00124 {  
00125     return node ? node->key : NULL;  
00126 }
```

4.23.1.5 hashnode_get_key_size()

```
size_t hashnode_get_key_size (  
    HashNode node )
```

Obtiene el tamaño de la clave del nodo.

Parameters

<i>node</i>	Puntero al nodo.
-------------	------------------

Returns

El tamaño de la clave del nodo.

Definition at line 162 of file [hashnode.c](#).

```
00162 {  
00163     return node->key_size;  
00164 }
```

4.23.1.6 hashnode_get_next()

```
HashNode hashnode_get_next (  
    HashNode node )
```

Obtiene el nodo siguiente.

Parameters

<i>node</i>	Puntero al nodo.
-------------	------------------

Returns

Puntero al nodo siguiente.

Definition at line 195 of file [hashnode.c](#).

```
00195 {  
00196     if (node == NULL) {
```

```

00197         return NULL;
00198     }
00199     return node->next;
00200 }

```

4.23.1.7 hashnode_get_prev()

```

HashNode hashnode_get_prev (
    HashNode node )

```

Obtiene el nodo previo.

Parameters

<i>node</i>	Puntero al nodo.
-------------	------------------

Returns

Puntero al nodo previo.

Definition at line 182 of file [hashnode.c](#).

```

00182                                     {
00183     if (node == NULL) {
00184         return NULL;
00185     }
00186     return node->prev;
00187 }

```

4.23.1.8 hashnode_get_prio()

```

LRUNode hashnode_get_prio (
    HashNode node )

```

Obtiene el puntero al nodo de prioridad asociado.

Parameters

<i>node</i>	Puntero al nodo objetivo.
-------------	---------------------------

Returns

Puntero a su nodo de prioridad.

Definition at line 208 of file [hashnode.c](#).

```

00208                                     {
00209     if (node == NULL) {
00210         return NULL;
00211     }
00212     return node->prio;
00213 }

```

4.23.1.9 hashnode_get_val()

```

void * hashnode_get_val (
    HashNode node )

```

Obtiene el valor del nodo.

Parameters

<i>node</i>	Puntero al nodo.
-------------	------------------

Returns

El puntero al valor del nodo.

Definition at line 143 of file [hashnode.c](#).

```
00143 {
00144     return node ? node->val : NULL;
00145 }
```

4.23.1.10 hashnode_get_val_size()

```
size_t hashnode_get_val_size (
    HashNode node )
```

Obtiene el tamaño del valor del nodo.

Parameters

<i>node</i>	Puntero al nodo.
-------------	------------------

Returns

El tamaño del valor del nodo.

Definition at line 172 of file [hashnode.c](#).

```
00172 {
00173     return node->val_size;
00174 }
```

4.23.1.11 hashnode_keys_equal()

```
int hashnode_keys_equal (
    const void * key_a,
    size_t size_a,
    const void * key_b,
    size_t size_b )
```

Definition at line 112 of file [hashnode.c](#).

```
00112 {
00113
00114     // PRINT("Comparacion: %s vs %s. Resultado: %i", key_a, key_b, size_a == size_b && memcmp(key_a,
key_b, size_a) == 0);
00115
00116     if (key_a == NULL || key_b == NULL)
00117         return 0;
00118
00119     return size_a == size_b && (memcmp(key_a, key_b, size_a) == 0);
00120
00121 }
```

4.23.1.12 hashnode_lookup()

```
LookupResult hashnode_lookup (
    void * key,
    size_t size,
    HashNode node )
```

Busca el valor asociado a la clave en el bucket iniciado en *node*.

Parameters

<i>key</i>	Clave a buscar.
<i>size</i>	El tamaño de la clave a buscar.
<i>node</i>	Bucket inicial.

Returns

Un [LookupResult](#) con el valor asociado y un status reflejando si la búsqueda fue exitosa, si no se encontro, o si se produjo algun error.

Definition at line 70 of file [hashnode.c](#).

```
00070                                     {
00071
00072     while (node) {
00073         if (hashnode_keys_equal(key, size, node->key, node->key_size))
00074             return create_ok_lookup_result(node->val, node->val_size);
00075
00076         node = node->next;
00077     }
00078
00079     return create_miss_lookup_result();
00080
00081 }
```

4.23.1.13 hashnode_lookup_node()

```
HashNode hashnode_lookup_node (
    void * key,
    size_t size,
    HashNode node )
```

Busca el puntero al nodo asociado a la clave en el bucket iniciado en *node*.

Parameters

<i>key</i>	Clave a buscar.
<i>size</i>	El tamaño de la clave a buscar.
<i>node</i>	Bucket inicial.

Returns

Un puntero al nodo buscado, que es NULL en caso de no encontrarse.

Definition at line 83 of file [hashnode.c](#).

```
00083                                     {
00084
```

```
00085     while (node) {
00086         if (hashnode_keys_equal(key, size, node->key, node->key_size))
00087             return node;
00088     }
00089     node = node->next;
00090 }
00091
00092 return NULL;
00093 }
```

4.23.1.14 hashnode_set_key()

```
int hashnode_set_key (
    HashNode node,
    void * key,
    size_t new_key_size )
```

Establece la clave del nodo, liberando la memoria de la clave anterior, en caso de existir.

Parameters

<i>node</i>	Puntero al nodo.
<i>key</i>	El puntero a la nueva clave para el nodo.
<i>new_key_size</i>	El tamaño de la nueva clave del nodo.

Returns

0 si la operacion es exitosa, -1 si no.

Definition at line 128 of file [hashnode.c](#).

```
00128
00129
00130     if (node == NULL)
00131         return -1;
00132
00133     if (node->key)
00134         free(node->key);
00135
00136     node->key = key;
00137     node->key_size = new_key_size;
00138
00139     return 0;
00140
00141 }
```

4.23.1.15 hashnode_set_key_size()

```
void hashnode_set_key_size (
    HashNode node,
    size_t key_size )
```

Establece el tamaño de la clave del nodo.

Parameters

<i>node</i>	Puntero al nodo.
<i>key_size</i>	El tamaño de la clave del nodo.

Definition at line 166 of file [hashnode.c](#).

```
00166                                     {
00167     if (node != NULL) {
00168         node->key_size = key_size;
00169     }
00170 }
```

4.23.1.16 hashnode_set_next()

```
void hashnode_set_next (
    HashNode node,
    HashNode next )
```

Establece el nodo siguiente.

Parameters

<i>node</i>	Puntero al nodo.
<i>next</i>	Puntero al nuevo nodo siguiente.

Definition at line 202 of file [hashnode.c](#).

```
00202                                     {
00203     if (node != NULL) {
00204         node->next = next;
00205     }
00206 }
```

4.23.1.17 hashnode_set_prev()

```
void hashnode_set_prev (
    HashNode node,
    HashNode prev )
```

Establece el nodo previo.

Parameters

<i>node</i>	Puntero al nodo.
<i>prev</i>	Puntero al nuevo nodo previo.

Definition at line 189 of file [hashnode.c](#).

```
00189                                     {
00190     if (node != NULL) {
00191         node->prev = prev;
00192     }
00193 }
```

4.23.1.18 hashnode_set_prio()

```
void hashnode_set_prio (
    HashNode node,
    LRUNode prio )
```

Establece el puntero al nodo de prioridad.

Parameters

<i>node</i>	Puntero al nodo objetivo.
<i>prio</i>	Puntero al nodo de prioridad.

Definition at line 215 of file [hashnode.c](#).

```
00215                                     {
00216     if (node != NULL) {
00217         node->prio = prio;
00218     }
00219 }
```

4.23.1.19 hashnode_set_val()

```
int hashnode_set_val (
    HashNode node,
    void * val,
    size_t new_val_size )
```

Establece la clave del nodo, liberando la memoria del valor anterior, en caso de existir.

Parameters

<i>node</i>	Puntero al nodo.
<i>val</i>	El puntero al nuevo valor para el nodo.
<i>new_val_size</i>	El tamaño del nuevo valor del nodo.

Returns

0 si la operacion es exitosa, -1 si no.

Definition at line 147 of file [hashnode.c](#).

```
00147                                     {
00148
00149     if (node == NULL)
00150         return -1;
00151
00152     if (node->val)
00153         free(node->val);
00154
00155     node->val = val;
00156     node->val_size = new_val_size;
00157
00158     return 0;
00159
00160 }
```

4.23.1.20 hashnode_set_val_size()

```
void hashnode_set_val_size (
    HashNode node,
    size_t val_size )
```

Establece el tamaño del valor del nodo.

Parameters

<i>node</i>	Puntero al nodo.
<i>val_size</i>	El tamaño del valor del nodo.

Definition at line 176 of file [hashnode.c](#).

```
00176                                     {
00177     if (node != NULL) {
00178         node->val_size = val_size;
00179     }
00180 }
```

4.24 hashnode.c

[Go to the documentation of this file.](#)

```
00001 #include <stdio.h>
00002 #include <string.h>
00003 #include <stdlib.h>
00004 #include <pthread.h>
00005 #include "hashnode.h"
00006 #include "../lru/lrunode.h"
00007
00008 struct HashNode {
00009     void* key;
00010     void* val;
00011     size_t key_size, val_size;
00012
00013     struct HashNode* prev;
00014     struct HashNode* next;
00015     struct LRUNode* prio;
00016 };
00017
00018
00019 int hashnode_keys_equal(const void* key_a, size_t size_a, const void* key_b, size_t size_b);
00020
00021
00022 HashNode hashnode_create(void* key, size_t key_size, void* val, size_t val_size, Cache cache) {
00023
00024     // No es posible insertar un par sin una key que lo identifique.
00025     if (key == NULL) return NULL;
00026
00027     HashNode node = dynalloc(sizeof(struct HashNode), cache);
00028     if (node == NULL)
00029         return NULL;
00030
00031     memset(node, 0, sizeof(struct HashNode));
00032
00033     LRUNode prio = lrunode_create(cache);
00034     if (prio == NULL) {
00035         free(node);
00036         return NULL;
00037     }
00038
00039     // Setteamos la clave
00040     node->key = key;
00041     node->key_size = key_size;
00042
00043     // Asignamos memoria y setteamos el valor
00044     node->val = val;
00045     node->val_size = val_size;
00046
00047     // Setteamos la prioridad
00048     node->prio = prio;
00049
00050     return node;
00051 }
00052
00053
00054 void hashnode_destroy(HashNode node) {
00055
00056     if (node == NULL)
00057         return;
00058
00059     // Liberamos la clave y el valor.
00060     if (node->key)
00061         free(node->key);
00062 }
```

```

00063     if (node->val)
00064         free(node->val);
00065
00066     free(node);
00067 }
00068 }
00069
00070 LookupResult hashnode_lookup(void* key, size_t size, HashNode node) {
00071
00072     while (node) {
00073         if (hashnode_keys_equal(key, size, node->key, node->key_size))
00074             return create_ok_lookup_result(node->val, node->val_size);
00075
00076         node = node->next;
00077     }
00078
00079     return create_miss_lookup_result();
00080 }
00081 }
00082
00083 HashNode hashnode_lookup_node(void* key, size_t size, HashNode node) {
00084
00085     while (node) {
00086         if (hashnode_keys_equal(key, size, node->key, node->key_size))
00087             return node;
00088
00089         node = node->next;
00090     }
00091
00092     return NULL;
00093 }
00094
00095 int hashnode_clean(HashNode node) {
00096
00097     if (node == NULL)
00098         return -1;
00099
00100     // Desconectamos y reconectamos los adyacentes
00101     HashNode prev = hashnode_get_prev(node);
00102     HashNode next = hashnode_get_next(node);
00103
00104     hashnode_set_next(prev, next);
00105     hashnode_set_prev(next, prev);
00106
00107     return 0;
00108 }
00109 }
00110
00111 // todo: ver como las vamos a comparar.
00112 int hashnode_keys_equal(const void* key_a, size_t size_a, const void* key_b, size_t size_b) {
00113
00114     // PRINT("Comparacion: %s vs %s. Resultado: %i", key_a, key_b, size_a == size_b && memcmp(key_a,
00115     key_b, size_a) == 0);
00116
00117     if (key_a == NULL || key_b == NULL)
00118         return 0;
00119
00120     return size_a == size_b && (memcmp(key_a, key_b, size_a) == 0);
00121 }
00122
00123
00124 void* hashnode_get_key(HashNode node) {
00125     return node ? node->key : NULL;
00126 }
00127
00128 int hashnode_set_key(HashNode node, void* key, size_t new_key_size) {
00129
00130     if (node == NULL)
00131         return -1;
00132
00133     if (node->key)
00134         free(node->key);
00135
00136     node->key = key;
00137     node->key_size = new_key_size;
00138
00139     return 0;
00140 }
00141 }
00142
00143 void* hashnode_get_val(HashNode node) {
00144     return node ? node->val : NULL;
00145 }
00146
00147 int hashnode_set_val(HashNode node, void* val, size_t new_val_size) {
00148

```

```

00149     if (node == NULL)
00150         return -1;
00151
00152     if (node->val)
00153         free(node->val);
00154
00155     node->val = val;
00156     node->val_size = new_val_size;
00157
00158     return 0;
00159 }
00160
00161 size_t hashnode_get_key_size(HashNode node) {
00162     return node->key_size;
00163 }
00164
00165 void hashnode_set_key_size(HashNode node, size_t key_size) {
00166     if (node != NULL) {
00167         node->key_size = key_size;
00168     }
00169 }
00170
00171 size_t hashnode_get_val_size(HashNode node) {
00172     return node->val_size;
00173 }
00174
00175 void hashnode_set_val_size(HashNode node, size_t val_size) {
00176     if (node != NULL) {
00177         node->val_size = val_size;
00178     }
00179 }
00180
00181 HashNode hashnode_get_prev(HashNode node) {
00182     if (node == NULL) {
00183         return NULL;
00184     }
00185     return node->prev;
00186 }
00187
00188 void hashnode_set_prev(HashNode node, HashNode prev) {
00189     if (node != NULL) {
00190         node->prev = prev;
00191     }
00192 }
00193
00194 HashNode hashnode_get_next(HashNode node) {
00195     if (node == NULL) {
00196         return NULL;
00197     }
00198     return node->next;
00199 }
00200
00201 void hashnode_set_next(HashNode node, HashNode next) {
00202     if (node != NULL) {
00203         node->next = next;
00204     }
00205 }
00206
00207 LRUNode hashnode_get_prio(HashNode node) {
00208     if (node == NULL) {
00209         return NULL;
00210     }
00211     return node->prio;
00212 }
00213
00214 void hashnode_set_prio(HashNode node, LRUNode prio) {
00215     if (node != NULL) {
00216         node->prio = prio;
00217     }
00218 }
00219 }

```

4.25 hashmap/hashnode.h File Reference

```

#include <stdio.h>
#include <pthread.h>
#include "../src/cache/cache.h"
#include "../dynalloc/dynalloc.h"
#include "../helpers/results.h"

```

Typedefs

- typedef struct [LRUNode](#) * [LRUNode](#)
- typedef struct [HashNode](#) * [HashNode](#)

Functions

- [HashNode](#) [hashnode_create](#) (void *key, size_t key_size, void *val, size_t val_size, [Cache](#) cache)
Crea un nuevo nodo.
- void [hashnode_destroy](#) ([HashNode](#) node)
Destruye el nodo objetivo, liberando la memoria asociada. Se asume que el nodo ya esta 'limpio'.
- [LookupResult](#) [hashnode_lookup](#) (void *key, size_t size, [HashNode](#) node)
Busca el valor asociado a la clave en el bucket iniciado en node.
- [HashNode](#) [hashnode_lookup_node](#) (void *key, size_t size, [HashNode](#) node)
Busca el puntero al nodo asociado a la clave en el bucket iniciado en node.
- int [hashnode_clean](#) ([HashNode](#) node)
Desconecta al nodo objetivo de sus vecinos, reconectando a sus nodos adyacentes.
- void * [hashnode_get_key](#) ([HashNode](#) node)
Obtiene la clave del nodo.
- int [hashnode_set_key](#) ([HashNode](#) node, void *key, size_t new_key_size)
Establece la clave del nodo, liberando la memoria de la clave anterior, en caso de existir.
- void * [hashnode_get_val](#) ([HashNode](#) node)
Obtiene el valor del nodo.
- int [hashnode_set_val](#) ([HashNode](#) node, void *val, size_t new_val_size)
Establece la clave del nodo, liberando la memoria del valor anterior, en caso de existir.
- size_t [hashnode_get_key_size](#) ([HashNode](#) node)
Obtiene el tamaño de la clave del nodo.
- void [hashnode_set_key_size](#) ([HashNode](#) node, size_t key_size)
Establece el tamaño de la clave del nodo.
- size_t [hashnode_get_val_size](#) ([HashNode](#) node)
Obtiene el tamaño del valor del nodo.
- void [hashnode_set_val_size](#) ([HashNode](#) node, size_t val_size)
Establece el tamaño del valor del nodo.
- [HashNode](#) [hashnode_get_prev](#) ([HashNode](#) node)
Obtiene el nodo previo.
- void [hashnode_set_prev](#) ([HashNode](#) node, [HashNode](#) prev)
Establece el nodo previo.
- [HashNode](#) [hashnode_get_next](#) ([HashNode](#) node)
Obtiene el nodo siguiente.
- void [hashnode_set_next](#) ([HashNode](#) node, [HashNode](#) next)
Establece el nodo siguiente.
- [LRUNode](#) [hashnode_get_prio](#) ([HashNode](#) node)
Obtiene el puntero al nodo de prioridad asociado.
- void [hashnode_set_prio](#) ([HashNode](#) node, [LRUNode](#) prio)
Establece el puntero al nodo de prioridad.

4.25.1 Typedef Documentation

4.25.1.1 HashNode

```
typedef struct HashNode* HashNode
```

Definition at line 13 of file [hashnode.h](#).

4.25.1.2 LRUNode

```
typedef struct LRUNode* LRUNode
```

Definition at line 11 of file [hashnode.h](#).

4.25.2 Function Documentation

4.25.2.1 hashnode_clean()

```
int hashnode_clean (
    HashNode node )
```

Desconecta al nodo objetivo de sus vecinos, reconectando a sus nodos adyacentes.

IMPORTANTE: Esta funcion NO es thread-safe. Debe pedirse el mutex asociado a la clave antes de invocarla, y liberarlo tras terminada la ejecucion.

Parameters

<i>node</i>	Nodo a limpiar.
-------------	-----------------

Returns

0 si es exitoso, 1 si se produjo un error.

Definition at line 95 of file [hashnode.c](#).

```
00095                                     {
00096
00097     if (node == NULL)
00098         return -1;
00099
00100     // Desconectamos y reconectamos los adyacentes
00101     HashNode prev = hashnode_get_prev(node);
00102     HashNode next = hashnode_get_next(node);
00103
00104     hashnode_set_next(prev, next);
00105     hashnode_set_prev(next, prev);
00106
00107     return 0;
00108
00109 }
```

4.25.2.2 hashnode_create()

```
HashNode hashnode_create (
    void * key,
    size_t key_size,
    void * val,
    size_t val_size,
    Cache cache )
```

Crea un nuevo nodo.

Parameters

<i>key</i>	El puntero a la clave del nodo.
<i>key_size</i>	El tamaño de la clave.
<i>val</i>	El puntero al valor del nodo.
<i>val_size</i>	El tamaño del valor.
<i>cache</i>	Puntero a la cache donde se insertara el nodo.

Returns

Un puntero al nodo creado, que es NULL si falla al asignarse memoria.

Definition at line 22 of file [hashnode.c](#).

```

00022                                     {
00023
00024     // No es posible insertar un par sin una key que lo identifique.
00025     if (key == NULL) return NULL;
00026
00027     HashNode node = dynalloc(sizeof(struct HashNode), cache);
00028     if (node == NULL)
00029         return NULL;
00030
00031     memset(node, 0, sizeof(struct HashNode));
00032
00033     LRUNode prio = lrnode_create(cache);
00034     if (prio == NULL) {
00035         free(node);
00036         return NULL;
00037     }
00038
00039     // Setteamos la clave
00040     node->key = key;
00041     node->key_size = key_size;
00042
00043     // Asignamos memoria y setteamos el valor
00044     node->val = val;
00045     node->val_size = val_size;
00046
00047     // Setteamos la prioridad
00048     node->prio = prio;
00049
00050     return node;
00051
00052 }
```

4.25.2.3 hashnode_destroy()

```

void hashnode_destroy (
    HashNode node )
```

Destruye el nodo objetivo, liberando la memoria asociada. Se asume que el nodo ya esta 'limpio'.

Parameters

<i>node</i>	El nodo a destruir.
-------------	---------------------

Definition at line 54 of file [hashnode.c](#).

```

00054                                     {
00055
00056     if (node == NULL)
00057         return;
00058
00059     // Liberamos la clave y el valor.
00060     if (node->key)
00061         free(node->key);
00062 }
```

```
00063     if (node->val)
00064         free(node->val);
00065
00066     free(node);
00067
00068 }
```

4.25.2.4 hashnode_get_key()

```
void * hashnode_get_key (
    HashNode node )
```

Obtiene la clave del nodo.

Parameters

<i>node</i>	Puntero al nodo.
-------------	------------------

Returns

El puntero a la clave del nodo.

Definition at line 124 of file [hashnode.c](#).

```
00124
00125     return node ? node->key : NULL;
00126 }
```

4.25.2.5 hashnode_get_key_size()

```
size_t hashnode_get_key_size (
    HashNode node )
```

Obtiene el tamaño de la clave del nodo.

Parameters

<i>node</i>	Puntero al nodo.
-------------	------------------

Returns

El tamaño de la clave del nodo.

Definition at line 162 of file [hashnode.c](#).

```
00162
00163     return node->key_size;
00164 }
```

4.25.2.6 hashnode_get_next()

```
HashNode hashnode_get_next (
    HashNode node )
```

Obtiene el nodo siguiente.

Parameters

<i>node</i>	Puntero al nodo.
-------------	------------------

Returns

Puntero al nodo siguiente.

Definition at line 195 of file [hashnode.c](#).

```
00195                                     {
00196     if (node == NULL) {
00197         return NULL;
00198     }
00199     return node->next;
00200 }
```

4.25.2.7 hashnode_get_prev()

```
HashNode hashnode_get_prev (
    HashNode node )
```

Obtiene el nodo previo.

Parameters

<i>node</i>	Puntero al nodo.
-------------	------------------

Returns

Puntero al nodo previo.

Definition at line 182 of file [hashnode.c](#).

```
00182                                     {
00183     if (node == NULL) {
00184         return NULL;
00185     }
00186     return node->prev;
00187 }
```

4.25.2.8 hashnode_get_prio()

```
LRUNode hashnode_get_prio (
    HashNode node )
```

Obtiene el puntero al nodo de prioridad asociado.

Parameters

<i>node</i>	Puntero al nodo objetivo.
-------------	---------------------------

Returns

Puntero a su nodo de prioridad.

Definition at line 208 of file [hashnode.c](#).

```
00208                                     {
00209     if (node == NULL) {
00210         return NULL;
00211     }
00212     return node->prio;
00213 }
```

4.25.2.9 hashnode_get_val()

```
void * hashnode_get_val (
    HashNode node )
```

Obtiene el valor del nodo.

Parameters

<i>node</i>	Puntero al nodo.
-------------	------------------

Returns

El puntero al valor del nodo.

Definition at line 143 of file [hashnode.c](#).

```
00143                                     {
00144     return node ? node->val : NULL;
00145 }
```

4.25.2.10 hashnode_get_val_size()

```
size_t hashnode_get_val_size (
    HashNode node )
```

Obtiene el tamaño del valor del nodo.

Parameters

<i>node</i>	Puntero al nodo.
-------------	------------------

Returns

El tamaño del valor del nodo.

Definition at line 172 of file [hashnode.c](#).

```
00172                                     {
00173     return node->val_size;
00174 }
```

4.25.2.11 hashnode_lookup()

```
LookupResult hashnode_lookup (
    void * key,
    size_t size,
    HashNode node )
```

Busca el valor asociado a la clave en el bucket iniciado en *node*.

Parameters

<i>key</i>	Clave a buscar.
<i>size</i>	El tamaño de la clave a buscar.
<i>node</i>	Bucket inicial.

Returns

Un [LookupResult](#) con el valor asociado y un status reflejando si la búsqueda fue exitosa, si no se encontro, o si se produjo algun error.

Definition at line 70 of file [hashnode.c](#).

```
00070                                     {
00071
00072     while (node) {
00073         if (hashnode_keys_equal(key, size, node->key, node->key_size))
00074             return create_ok_lookup_result(node->val, node->val_size);
00075
00076         node = node->next;
00077     }
00078
00079     return create_miss_lookup_result();
00080
00081 }
```

4.25.2.12 hashnode_lookup_node()

```
HashNode hashnode_lookup_node (
    void * key,
    size_t size,
    HashNode node )
```

Busca el puntero al nodo asociado a la clave en el bucket iniciado en *node*.

Parameters

<i>key</i>	Clave a buscar.
<i>size</i>	El tamaño de la clave a buscar.
<i>node</i>	Bucket inicial.

Returns

Un puntero al nodo buscado, que es NULL en caso de no encontrarse.

Definition at line 83 of file [hashnode.c](#).

```
00083                                     {
00084
00085     while (node) {
00086         if (hashnode_keys_equal(key, size, node->key, node->key_size))
00087             return node;
00088
00089         node = node->next;
00090     }
00091
00092     return NULL;
00093 }
```

4.25.2.13 hashnode_set_key()

```
int hashnode_set_key (
    HashNode node,
    void * key,
    size_t new_key_size )
```

Establece la clave del nodo, liberando la memoria de la clave anterior, en caso de existir.

Parameters

<i>node</i>	Puntero al nodo.
<i>key</i>	El puntero a la nueva clave para el nodo.
<i>new_key_size</i>	El tamaño de la nueva clave del nodo.

Returns

0 si la operacion es exitosa, -1 si no.

Definition at line 128 of file [hashnode.c](#).

```
00128                                     {
00129
00130     if (node == NULL)
00131         return -1;
00132
00133     if (node->key)
00134         free(node->key);
00135
00136     node->key = key;
00137     node->key_size = new_key_size;
00138
00139     return 0;
00140
00141 }
```

4.25.2.14 hashnode_set_key_size()

```
void hashnode_set_key_size (
    HashNode node,
    size_t key_size )
```

Establece el tamaño de la clave del nodo.

Parameters

<i>node</i>	Puntero al nodo.
<i>key_size</i>	El tamaño de la clave del nodo.

Definition at line 166 of file [hashnode.c](#).

```
00166                                     {
00167     if (node != NULL) {
00168         node->key_size = key_size;
00169     }
00170 }
```

4.25.2.15 hashnode_set_next()

```
void hashnode_set_next (
```

```
    HashNode node,  
    HashNode next )
```

Establece el nodo siguiente.

Parameters

<i>node</i>	Puntero al nodo.
<i>next</i>	Puntero al nuevo nodo siguiente.

Definition at line 202 of file [hashnode.c](#).

```
00202                                     {  
00203     if (node != NULL) {  
00204         node->next = next;  
00205     }  
00206 }
```

4.25.2.16 hashnode_set_prev()

```
void hashnode_set_prev (  
    HashNode node,  
    HashNode prev )
```

Establece el nodo previo.

Parameters

<i>node</i>	Puntero al nodo.
<i>prev</i>	Puntero al nuevo nodo previo.

Definition at line 189 of file [hashnode.c](#).

```
00189                                     {  
00190     if (node != NULL) {  
00191         node->prev = prev;  
00192     }  
00193 }
```

4.25.2.17 hashnode_set_prio()

```
void hashnode_set_prio (  
    HashNode node,  
    LRUNode prio )
```

Establece el puntero al nodo de prioridad.

Parameters

<i>node</i>	Puntero al nodo objetivo.
<i>prio</i>	Puntero al nodo de prioridad.

Definition at line 215 of file [hashnode.c](#).

```
00215                                     {  
00216     if (node != NULL) {  
00217         node->prio = prio;  
00218     }  
00219 }
```

```
00218     }
00219 }
```

4.25.2.18 hashnode_set_val()

```
int hashnode_set_val (
    HashNode node,
    void * val,
    size_t new_val_size )
```

Establece la clave del nodo, liberando la memoria del valor anterior, en caso de existir.

Parameters

<i>node</i>	Puntero al nodo.
<i>val</i>	El puntero al nuevo valor para el nodo.
<i>new_val_size</i>	El tamaño del nuevo valor del nodo.

Returns

0 si la operacion es exitosa, -1 si no.

Definition at line 147 of file [hashnode.c](#).

```
00147                                     {
00148
00149     if (node == NULL)
00150         return -1;
00151
00152     if (node->val)
00153         free(node->val);
00154
00155     node->val = val;
00156     node->val_size = new_val_size;
00157
00158     return 0;
00159
00160 }
```

4.25.2.19 hashnode_set_val_size()

```
void hashnode_set_val_size (
    HashNode node,
    size_t val_size )
```

Establece el tamaño del valor del nodo.

Parameters

<i>node</i>	Puntero al nodo.
<i>val_size</i>	El tamaño del valor del nodo.

Definition at line 176 of file [hashnode.c](#).

```
00176                                     {
00177     if (node != NULL) {
00178         node->val_size = val_size;
00179     }
00180 }
```

4.26 hashnode.h

[Go to the documentation of this file.](#)

```

00001 #ifndef __HASH_NODE_H__
00002 #define __HASH_NODE_H__
00003
00004 #include <stdio.h>
00005 #include <pthread.h>
00006 #include "../src/cache/cache.h"
00007 #include "../dynalloc/dynalloc.h"
00008 #include "../helpers/results.h"
00009
00010 // Forward-declaration de LRUNode para evitar incluirla
00011 typedef struct LRUNode* LRUNode;
00012
00013 typedef struct HashNode* HashNode;
00014
00015
00016 /**
00017  * @brief Crea un nuevo nodo.
00018  *
00019  * @param key El puntero a la clave del nodo.
00020  * @param key_size El tamaño de la clave.
00021  * @param val El puntero al valor del nodo.
00022  * @param val_size El tamaño del valor.
00023  * @param cache Puntero a la cache donde se insertara el nodo.
00024  *
00025  * @return Un puntero al nodo creado, que es NULL si falla al asignarse memoria.
00026  */
00027 HashNode hashnode_create(void* key, size_t key_size, void* val, size_t val_size, Cache cache);
00028
00029
00030 /**
00031  * @brief Destruye el nodo objetivo, liberando la memoria asociada. Se asume que el nodo ya esta
00032  * 'limpio'.
00033  *
00034  * @param node El nodo a destruir.
00035  */
00036 void hashnode_destroy(HashNode node);
00037
00038 /**
00039  * @brief Busca el valor asociado a la clave en el bucket iniciado en \a node.
00040  *
00041  * @param key Clave a buscar.
00042  * @param size El tamaño de la clave a buscar.
00043  * @param node Bucket inicial.
00044  * @return Un \a LookupResult con el valor asociado y un status reflejando si la búsqueda fue exitosa,
00045  * * si no se encontro, o si se produjo algun error.
00046  */
00047 LookupResult hashnode_lookup(void* key, size_t size, HashNode node);
00048
00049
00050 /**
00051  * @brief Busca el puntero al nodo asociado a la clave en el bucket iniciado en \a node.
00052  *
00053  * @param key Clave a buscar.
00054  * @param size El tamaño de la clave a buscar.
00055  * @param node Bucket inicial.
00056  * @return Un puntero al nodo buscado, que es NULL en caso de no encontrarse.
00057  */
00058 HashNode hashnode_lookup_node(void* key, size_t size, HashNode node);
00059
00060 /**
00061  * @brief Desconecta al nodo objetivo de sus vecinos, reconectando a sus nodos adyacentes.
00062  *
00063  * IMPORTANTE: Esta funcion NO es thread-safe. Debe pedirse el mutex asociado a la clave antes de
00064  * invocarla, y
00065  * liberarlo tras terminada la ejecucion.
00066  *
00067  * @param node Nodo a limpiar.
00068  * @return 0 si es exitoso, 1 si se produjo un error.
00069  */
00070 int hashnode_clean(HashNode node);
00071
00072
00073
00074 // ! Obs: los setters para key y val van a tener que tomar a Cache como parametro para poder llamar a
00075 dynalloc.
00076
00077 /**
00078  * @brief Obtiene la clave del nodo.
00079  *

```

```

00080  * @param node Puntero al nodo.
00081  * @return El puntero a la clave del nodo.
00082  */
00083 void* hashnode_get_key(HashNode node);
00084
00085 /**
00086  * @brief Establece la clave del nodo, liberando la memoria de la clave anterior, en caso de existir.
00087  *
00088  * @param node Puntero al nodo.
00089  * @param key El puntero a la nueva clave para el nodo.
00090  * @param new_key_size El tamaño de la nueva clave del nodo.
00091  *
00092  * @return 0 si la operacion es exitosa, -1 si no.
00093  */
00094 int hashnode_set_key(HashNode node, void* key, size_t new_key_size);
00095
00096
00097 /**
00098  * @brief Obtiene el valor del nodo.
00099  *
00100  * @param node Puntero al nodo.
00101  * @return El puntero al valor del nodo.
00102  */
00103 void* hashnode_get_val(HashNode node);
00104
00105
00106 /**
00107  * @brief Establece la clave del nodo, liberando la memoria del valor anterior, en caso de existir.
00108  *
00109  * @param node Puntero al nodo.
00110  * @param val El puntero al nuevo valor para el nodo.
00111  * @param new_val_size El tamaño del nuevo valor del nodo.
00112  *
00113  * @return 0 si la operacion es exitosa, -1 si no.
00114  */
00115 int hashnode_set_val(HashNode node, void* val, size_t new_val_size);
00116
00117 /**
00118  * @brief Obtiene el tamaño de la clave del nodo.
00119  *
00120  * @param node Puntero al nodo.
00121  * @return El tamaño de la clave del nodo.
00122  */
00123 size_t hashnode_get_key_size(HashNode node);
00124
00125
00126 /**
00127  * @brief Establece el tamaño de la clave del nodo.
00128  *
00129  * @param node Puntero al nodo.
00130  * @param key_size El tamaño de la clave del nodo.
00131  */
00132 void hashnode_set_key_size(HashNode node, size_t key_size);
00133
00134
00135 /**
00136  * @brief Obtiene el tamaño del valor del nodo.
00137  *
00138  * @param node Puntero al nodo.
00139  * @return El tamaño del valor del nodo.
00140  */
00141 size_t hashnode_get_val_size(HashNode node);
00142
00143
00144 /**
00145  * @brief Establece el tamaño del valor del nodo.
00146  *
00147  * @param node Puntero al nodo.
00148  * @param val_size El tamaño del valor del nodo.
00149  */
00150 void hashnode_set_val_size(HashNode node, size_t val_size);
00151
00152
00153 /**
00154  * @brief Obtiene el nodo previo.
00155  * @param node Puntero al nodo.
00156  * @return Puntero al nodo previo.
00157  */
00158 HashNode hashnode_get_prev(HashNode node);
00159
00160 /**
00161  * @brief Establece el nodo previo.
00162  *
00163  * @param node Puntero al nodo.
00164  * @param prev Puntero al nuevo nodo previo.
00165  */
00166 void hashnode_set_prev(HashNode node, HashNode prev);

```



```

00167
00168 /**
00169  * @brief Obtiene el nodo siguiente.
00170  *
00171  * @param node Puntero al nodo.
00172  * @return Puntero al nodo siguiente.
00173  */
00174 HashNode hashnode_get_next (HashNode node);
00175
00176 /**
00177  * @brief Establece el nodo siguiente.
00178  *
00179  * @param node Puntero al nodo.
00180  * @param next Puntero al nuevo nodo siguiente.
00181  */
00182 void hashnode_set_next (HashNode node, HashNode next);
00183
00184 /**
00185  * @brief Obtiene el puntero al nodo de prioridad asociado.
00186  *
00187  * @param node Puntero al nodo objetivo.
00188  * @return Puntero a su nodo de prioridad.
00189  */
00190 LRUNode hashnode_get_prio (HashNode node);
00191
00192 /**
00193  * @brief Establece el puntero al nodo de prioridad.
00194  * @param node Puntero al nodo objetivo.
00195  * @param prio Puntero al nodo de prioridad.
00196  */
00197 void hashnode_set_prio (HashNode node, LRUNode prio);
00198
00199 #endif // __HASH_NODE_H__

```

4.27 helpers/atom_counter.c File Reference

```

#include <stdlib.h>
#include <pthread.h>
#include "atom_counter.h"

```

Classes

- struct [AtomCounter](#)

Functions

- [AtomCounter atom_counter_create](#) (unsigned int initial_value)
- [Counter atom_counter_get](#) ([AtomCounter](#) counter)
- int [atom_counter_inc](#) ([AtomCounter](#) counter)
- int [atom_counter_dec](#) ([AtomCounter](#) counter)
- int [atom_counter_add](#) ([AtomCounter](#) counter, [Counter](#) n)
- int [atom_counter_drop](#) ([AtomCounter](#) counter, [Counter](#) n)
- int [atom_counter_destroy](#) ([AtomCounter](#) counter)

4.27.1 Function Documentation

4.27.1.1 atom_counter_add()

```
int atom_counter_add (
    AtomCounter counter,
    Counter n )
```

Definition at line 82 of file [atom_counter.c](#).

```
00082                                     {
00083
00084     if (counter == NULL)
00085         return -1;
00086
00087     pthread_rwlock_wrlock(&counter->lock);
00088
00089     counter->counter += n;
00090
00091     pthread_rwlock_unlock(&counter->lock);
00092
00093     return 0;
00094
00095 }
```

4.27.1.2 atom_counter_create()

```
AtomCounter atom_counter_create (
    unsigned int initial_value )
```

Definition at line 14 of file [atom_counter.c](#).

```
00014                                     {
00015
00016     AtomCounter atom_counter = malloc(sizeof(struct AtomCounter));
00017
00018     if (atom_counter == NULL)
00019         return NULL;
00020
00021     atom_counter->counter = initial_value;
00022
00023     if (pthread_rwlock_init(&atom_counter->lock, NULL) != 0) {
00024         free(atom_counter);
00025         return NULL;
00026     }
00027
00028     return atom_counter;
00029
00030 }
```

4.27.1.3 atom_counter_dec()

```
int atom_counter_dec (
    AtomCounter counter )
```

Definition at line 65 of file [atom_counter.c](#).

```
00065                                     {
00066
00067     if (counter == NULL)
00068         return -1;
00069
00070     pthread_rwlock_wrlock(&counter->lock);
00071
00072     if (counter->counter > 0)
00073         counter->counter--;
00074
00075     pthread_rwlock_unlock(&counter->lock);
00076
00077     return 0;
00078
00079 }
```

4.27.1.4 atom_counter_destroy()

```
int atom_counter_destroy (
    AtomCounter counter )
```

Definition at line 117 of file [atom_counter.c](#).

```
00117 {
00118
00119     if (counter == NULL)
00120         return -1;
00121
00122     pthread_rwlock_destroy(&counter->lock);
00123
00124     free(counter);
00125
00126     return 0;
00127 }
00128 }
```

4.27.1.5 atom_counter_drop()

```
int atom_counter_drop (
    AtomCounter counter,
    Counter n )
```

Definition at line 98 of file [atom_counter.c](#).

```
00098 {
00099
00100     if (counter == NULL)
00101         return -1;
00102
00103     pthread_rwlock_wrlock(&counter->lock);
00104
00105     if (counter->counter > n)
00106         counter->counter -= n;
00107     else
00108         counter->counter = 0;
00109
00110     pthread_rwlock_unlock(&counter->lock);
00111
00112     return 0;
00113 }
00114 }
```

4.27.1.6 atom_counter_get()

```
Counter atom_counter_get (
    AtomCounter counter )
```

Definition at line 33 of file [atom_counter.c](#).

```
00033 {
00034
00035     if (counter == NULL)
00036         return 0;
00037
00038     pthread_rwlock_rdlock(&counter->lock);
00039
00040     Counter count = counter->counter;
00041
00042     pthread_rwlock_unlock(&counter->lock);
00043
00044     return count;
00045 }
00046 }
```

4.27.1.7 atom_counter_inc()

```
int atom_counter_inc (
    AtomCounter counter )
```

Definition at line 49 of file [atom_counter.c](#).

```
00049 {
00050
00051     if (counter == NULL)
00052         return -1;
00053
00054     pthread_rwlock_wrlock(&counter->lock);
00055
00056     counter->counter++;
00057
00058     pthread_rwlock_unlock(&counter->lock);
00059
00060     return 0;
00061
00062 }
```

4.28 atom_counter.c

[Go to the documentation of this file.](#)

```
00001 #include <stdlib.h>
00002 #include <pthread.h>
00003 #include "atom_counter.h"
00004
00005 struct AtomCounter {
00006     Counter counter;
00007
00008     pthread_rwlock_t lock;
00009
00010 };
00011
00012
00013
00014 AtomCounter atom_counter_create(unsigned int initial_value) {
00015     AtomCounter atom_counter = malloc(sizeof(struct AtomCounter));
00016
00017     if (atom_counter == NULL)
00018         return NULL;
00019
00020     atom_counter->counter = initial_value;
00021
00022     if (pthread_rwlock_init(&atom_counter->lock, NULL) != 0) {
00023         free(atom_counter);
00024         return NULL;
00025     }
00026
00027     return atom_counter;
00028
00029 }
00030
00031
00032
00033 Counter atom_counter_get(AtomCounter counter) {
00034
00035     if (counter == NULL)
00036         return 0;
00037
00038     pthread_rwlock_rdlock(&counter->lock);
00039
00040     Counter count = counter->counter;
00041
00042     pthread_rwlock_unlock(&counter->lock);
00043
00044     return count;
00045
00046 }
00047
00048
00049 int atom_counter_inc(AtomCounter counter) {
00050
00051     if (counter == NULL)
00052         return -1;
00053
00054     pthread_rwlock_wrlock(&counter->lock);
```

```

00055
00056     counter->counter++;
00057
00058     pthread_rwlock_unlock(&counter->lock);
00059
00060     return 0;
00061 }
00062 }
00063
00064
00065 int atom_counter_dec(AtomCounter counter) {
00066
00067     if (counter == NULL)
00068         return -1;
00069
00070     pthread_rwlock_wrlock(&counter->lock);
00071
00072     if (counter->counter > 0)
00073         counter->counter--;
00074
00075     pthread_rwlock_unlock(&counter->lock);
00076
00077     return 0;
00078 }
00079 }
00080
00081
00082 int atom_counter_add(AtomCounter counter, Counter n) {
00083
00084     if (counter == NULL)
00085         return -1;
00086
00087     pthread_rwlock_wrlock(&counter->lock);
00088
00089     counter->counter += n;
00090
00091     pthread_rwlock_unlock(&counter->lock);
00092
00093     return 0;
00094 }
00095 }
00096
00097
00098 int atom_counter_drop(AtomCounter counter, Counter n) {
00099
00100     if (counter == NULL)
00101         return -1;
00102
00103     pthread_rwlock_wrlock(&counter->lock);
00104
00105     if (counter->counter > n)
00106         counter->counter -= n;
00107     else
00108         counter->counter = 0;
00109
00110     pthread_rwlock_unlock(&counter->lock);
00111
00112     return 0;
00113 }
00114 }
00115
00116
00117 int atom_counter_destroy(AtomCounter counter) {
00118
00119     if (counter == NULL)
00120         return -1;
00121
00122     pthread_rwlock_destroy(&counter->lock);
00123
00124     free(counter);
00125
00126     return 0;
00127 }
00128 }

```

4.29 helpers/atom_counter.h File Reference

Macros

- #define COUNTER_FORMAT "%lu"

Typedefs

- typedef `__uint64_t` [Counter](#)
- typedef struct [AtomCounter](#) * [AtomCounter](#)

Functions

- [AtomCounter](#) [atom_counter_create](#) (unsigned int initial_value)
- [Counter](#) [atom_counter_get](#) ([AtomCounter](#) counter)
- int [atom_counter_inc](#) ([AtomCounter](#) counter)
- int [atom_counter_dec](#) ([AtomCounter](#) counter)
- int [atom_counter_add](#) ([AtomCounter](#) counter, [Counter](#) n)
- int [atom_counter_drop](#) ([AtomCounter](#) counter, [Counter](#) n)
- int [atom_counter_destroy](#) ([AtomCounter](#) counter)

4.29.1 Macro Definition Documentation

4.29.1.1 COUNTER_FORMAT

```
#define COUNTER_FORMAT "%lu"
```

Definition at line 4 of file [atom_counter.h](#).

4.29.2 Typedef Documentation

4.29.2.1 AtomCounter

```
typedef struct AtomCounter* AtomCounter
```

Definition at line 7 of file [atom_counter.h](#).

4.29.2.2 Counter

```
typedef __uint64_t Counter
```

Definition at line 5 of file [atom_counter.h](#).

4.29.3 Function Documentation

4.29.3.1 atom_counter_add()

```
int atom_counter_add (
    AtomCounter counter,
    Counter n )
```

Definition at line 82 of file [atom_counter.c](#).

```
00082                                     {
00083
00084     if (counter == NULL)
00085         return -1;
00086
00087     pthread_rwlock_wrlock(&counter->lock);
00088
00089     counter->counter += n;
00090
00091     pthread_rwlock_unlock(&counter->lock);
00092
00093     return 0;
00094
00095 }
```

4.29.3.2 atom_counter_create()

```
AtomCounter atom_counter_create (  
    unsigned int initial_value )
```

Definition at line 14 of file [atom_counter.c](#).

```
00014                                     {  
00015  
00016     AtomCounter atom_counter = malloc(sizeof(struct AtomCounter));  
00017  
00018     if (atom_counter == NULL)  
00019         return NULL;  
00020  
00021     atom_counter->counter = initial_value;  
00022  
00023     if (pthread_rwlock_init(&atom_counter->lock, NULL) != 0) {  
00024         free(atom_counter);  
00025         return NULL;  
00026     }  
00027  
00028     return atom_counter;  
00029  
00030 }
```

4.29.3.3 atom_counter_dec()

```
int atom_counter_dec (  
    AtomCounter counter )
```

Definition at line 65 of file [atom_counter.c](#).

```
00065                                     {  
00066  
00067     if (counter == NULL)  
00068         return -1;  
00069  
00070     pthread_rwlock_wrlock(&counter->lock);  
00071  
00072     if (counter->counter > 0)  
00073         counter->counter--;  
00074  
00075     pthread_rwlock_unlock(&counter->lock);  
00076  
00077     return 0;  
00078  
00079 }
```

4.29.3.4 atom_counter_destroy()

```
int atom_counter_destroy (  
    AtomCounter counter )
```

Definition at line 117 of file [atom_counter.c](#).

```
00117                                     {  
00118  
00119     if (counter == NULL)  
00120         return -1;  
00121  
00122     pthread_rwlock_destroy(&counter->lock);  
00123  
00124     free(counter);  
00125  
00126     return 0;  
00127  
00128 }
```

4.29.3.5 atom_counter_drop()

```
int atom_counter_drop (
    AtomCounter counter,
    Counter n )
```

Definition at line 98 of file [atom_counter.c](#).

```
00098                                     {
00099
00100     if (counter == NULL)
00101         return -1;
00102
00103     pthread_rwlock_wrlock(&counter->lock);
00104
00105     if (counter->counter > n)
00106         counter->counter -= n;
00107     else
00108         counter->counter = 0;
00109
00110     pthread_rwlock_unlock(&counter->lock);
00111
00112     return 0;
00113
00114 }
```

4.29.3.6 atom_counter_get()

```
Counter atom_counter_get (
    AtomCounter counter )
```

Definition at line 33 of file [atom_counter.c](#).

```
00033                                     {
00034
00035     if (counter == NULL)
00036         return 0;
00037
00038     pthread_rwlock_rdlock(&counter->lock);
00039
00040     Counter count = counter->counter;
00041
00042     pthread_rwlock_unlock(&counter->lock);
00043
00044     return count;
00045
00046 }
```

4.29.3.7 atom_counter_inc()

```
int atom_counter_inc (
    AtomCounter counter )
```

Definition at line 49 of file [atom_counter.c](#).

```
00049                                     {
00050
00051     if (counter == NULL)
00052         return -1;
00053
00054     pthread_rwlock_wrlock(&counter->lock);
00055
00056     counter->counter++;
00057
00058     pthread_rwlock_unlock(&counter->lock);
00059
00060     return 0;
00061
00062 }
```


4.30 atom_counter.h

[Go to the documentation of this file.](#)

```
00001 #ifndef __ATOM_COUNTER_H__
00002 #define __ATOM_COUNTER_H__
00003
00004 #define COUNTER_FORMAT "%lu"
00005 typedef __uint64_t Counter;
00006
00007 typedef struct AtomCounter* AtomCounter;
00008
00009
00010 AtomCounter atom_counter_create(unsigned int initial_value);
00011
00012 Counter atom_counter_get(AtomCounter counter);
00013
00014 int atom_counter_inc(AtomCounter counter);
00015
00016 int atom_counter_dec(AtomCounter counter);
00017
00018 int atom_counter_add(AtomCounter counter, Counter n);
00019
00020 int atom_counter_drop(AtomCounter counter, Counter n);
00021
00022 int atom_counter_destroy(AtomCounter counter);
00023
00024 #endif // __ATOMIC_COUNTER_H__
```

4.31 helpers/quit.c File Reference

```
#include <stdlib.h>
#include <stdio.h>
#include "quit.h"
```

Functions

- void [quit](#) (char *error)

Imprime el mensaje error explicando el valor de errno y aborta la ejecucion generando un core dump al invocar a abort().

4.31.1 Function Documentation

4.31.1.1 quit()

```
void quit (
    char * error )
```

Imprime el mensaje error explicando el valor de errno y aborta la ejecucion generando un core dump al invocar a abort().

Parameters

error	Mensaje de error explicando el valor de error de errno.
-------	---

Definition at line 5 of file [quit.c](#).

```
00005 {
```

```
00006    perror(error);
00007    abort();
00008 }
```

4.32 quit.c

[Go to the documentation of this file.](#)

```
00001 #include <stdlib.h>
00002 #include <stdio.h>
00003 #include "quit.h"
00004
00005 void quit(char* error) {
00006     perror(error);
00007     abort();
00008 }
```

4.33 helpers/quit.h File Reference

Functions

- void `quit` (char *error)

Imprime el mensaje `error` explicando el valor de `errno` y aborta la ejecucion generando un core dump al invocar a `abort()`.

4.33.1 Function Documentation

4.33.1.1 quit()

```
void quit (
    char * error )
```

Imprime el mensaje `error` explicando el valor de `errno` y aborta la ejecucion generando un core dump al invocar a `abort()`.

Parameters

<i>error</i>	Mensaje de error explicando el valor de error de <code>errno</code> .
--------------	---

Definition at line 5 of file `quit.c`.

```
00005 {
00006     perror(error);
00007     abort();
00008 }
```

4.34 quit.h

[Go to the documentation of this file.](#)

```
00001 #ifndef __QUIT_H__
00002 #define __QUIT_H__
00003
00004 /**
```

```

00005  * @brief Imprime el mensaje `error` explicando el valor de `errno` y aborta la ejecucion generando
        un core dump al invocar a `abort()`.
00006  *
00007  * @param error Mensaje de error explicando el valor de error de errno.
00008  */
00009 void quit(char* error);
00010
00011 #endif // __QUIT_H__

```

4.35 helpers/results.c File Reference

```

#include <stdlib.h>
#include "results.h"

```

Functions

- [LookupResult create_ok_lookup_result](#) (void *ptr, size_t size)
Devuelve un [LookupResult](#) con el valor objetivo y status OK.
- [LookupResult create_error_lookup_result](#) ()
Devuelve un [LookupResult](#) con valor 0 y status ERROR.
- [LookupResult create_miss_lookup_result](#) ()
Devuelve un [LookupResult](#) con valor 0 y status MISS.
- int [lookup_result_is_error](#) (LookupResult lr)
Devuelve 1 si el status del [LookupResult](#) objetivo es ERROR, o 0 en caso contrario.
- int [lookup_result_is_ok](#) (LookupResult lr)
Devuelve 1 si el status del [LookupResult](#) objetivo es OK, o 0 en caso contrario.
- int [lookup_result_is_miss](#) (LookupResult lr)
Devuelve 1 si el status del [LookupResult](#) objetivo es MISS, o 0 en caso contrario.
- void * [lookup_result_get_value](#) (LookupResult lr)
Devuelve el valor almacenado en el [LookupResult](#).
- size_t [lookup_result_get_size](#) (LookupResult lr)
Devuelve el tamaño del valor almacenado en el [LookupResult](#).

4.35.1 Function Documentation

4.35.1.1 create_error_lookup_result()

```
LookupResult create_error_lookup_result ( ) [inline]
```

Devuelve un [LookupResult](#) con valor 0 y status ERROR.

Definition at line 12 of file [results.c](#).

```

00012 {
00013     return create_lookup_result(NULL, 0, ERROR);
00014 }

```

4.35.1.2 create_miss_lookup_result()

```
LookupResult create_miss_lookup_result ( ) [inline]
```

Devuelve un [LookupResult](#) con valor 0 y status MISS.

Definition at line 16 of file [results.c](#).

```

00016 {
00017     return create_lookup_result(NULL, 0, MISS);
00018 }

```

4.35.1.3 create_ok_lookup_result()

```
LookupResult create_ok_lookup_result (
    void * ptr,
    size_t size ) [inline]
```

Devuelve un [LookupResult](#) con el valor objetivo y status OK.

Definition at line 8 of file [results.c](#).

```
00008                                     {
00009     return create_lookup_result(ptr, size, OK);
00010 }
```

4.35.1.4 lookup_result_get_size()

```
size_t lookup_result_get_size (
    LookupResult lr )
```

Devuelve el tamaño del valor almacenado en el [LookupResult](#).

Definition at line 36 of file [results.c](#).

```
00036                                     {
00037     return lr.size;
00038 }
```

4.35.1.5 lookup_result_get_value()

```
void * lookup_result_get_value (
    LookupResult lr )
```

Devuelve el valor almacenado en el [LookupResult](#).

Definition at line 32 of file [results.c](#).

```
00032                                     {
00033     return lr.ptr;
00034 }
```

4.35.1.6 lookup_result_is_error()

```
int lookup_result_is_error (
    LookupResult lr ) [inline]
```

Devuelve 1 si el status del [LookupResult](#) objetivo es ERROR, o 0 en caso contrario.

Definition at line 20 of file [results.c](#).

```
00020                                     {
00021     return lr.status == ERROR;
00022 }
```

4.35.1.7 lookup_result_is_miss()

```
int lookup_result_is_miss (
    LookupResult lr ) [inline]
```

Devuelve 1 si el status del [LookupResult](#) objetivo es MISS, o 0 en caso contrario.

Definition at line 28 of file [results.c](#).

```
00028                                     {
00029     return lr.status == MISS;
00030 }
```

4.35.1.8 lookup_result_is_ok()

```
int lookup_result_is_ok (
    LookupResult lr ) [inline]
```

Devuelve 1 si el status del [LookupResult](#) objetivo es OK, o 0 en caso contrario.

Definition at line 24 of file [results.c](#).

```
00024 {
00025     return lr.status == OK;
00026 }
```

4.36 results.c

[Go to the documentation of this file.](#)

```
00001 #include <stdlib.h>
00002 #include "results.h"
00003
00004 static inline LookupResult create_lookup_result(void* ptr, size_t size, Status status) {
00005     return (LookupResult){ptr, size, status};
00006 }
00007
00008 inline LookupResult create_ok_lookup_result(void* ptr, size_t size) {
00009     return create_lookup_result(ptr, size, OK);
00010 }
00011
00012 inline LookupResult create_error_lookup_result() {
00013     return create_lookup_result(NULL, 0, ERROR);
00014 }
00015
00016 inline LookupResult create_miss_lookup_result() {
00017     return create_lookup_result(NULL, 0, MISS);
00018 }
00019
00020 inline int lookup_result_is_error(LookupResult lr) {
00021     return lr.status == ERROR;
00022 }
00023
00024 inline int lookup_result_is_ok(LookupResult lr) {
00025     return lr.status == OK;
00026 }
00027
00028 inline int lookup_result_is_miss(LookupResult lr) {
00029     return lr.status == MISS;
00030 }
00031
00032 void* lookup_result_get_value(LookupResult lr) {
00033     return lr.ptr;
00034 }
00035
00036 size_t lookup_result_get_size(LookupResult lr) {
00037     return lr.size;
00038 }
00039
```

4.37 helpers/results.h File Reference

Classes

- struct [LookupResult](#)

Typedefs

- typedef struct [LookupResult](#) [LookupResult](#)

Enumerations

- enum [Status](#) { [OK](#) , [ERROR](#) , [MISS](#) }

Functions

- [LookupResult create_ok_lookup_result](#) (void *ptr, size_t size)
Devuelve un [LookupResult](#) con el valor objetivo y status OK.
- [LookupResult create_error_lookup_result](#) ()
Devuelve un [LookupResult](#) con valor 0 y status ERROR.
- [LookupResult create_miss_lookup_result](#) ()
Devuelve un [LookupResult](#) con valor 0 y status MISS.
- int [lookup_result_is_error](#) ([LookupResult](#) lr)
Devuelve 1 si el status del [LookupResult](#) objetivo es ERROR, o 0 en caso contrario.
- int [lookup_result_is_ok](#) ([LookupResult](#) lr)
Devuelve 1 si el status del [LookupResult](#) objetivo es OK, o 0 en caso contrario.
- int [lookup_result_is_miss](#) ([LookupResult](#) lr)
Devuelve 1 si el status del [LookupResult](#) objetivo es MISS, o 0 en caso contrario.
- void * [lookup_result_get_value](#) ([LookupResult](#) lr)
Devuelve el valor almacenado en el [LookupResult](#).
- size_t [lookup_result_get_size](#) ([LookupResult](#) lr)
Devuelve el tamaño del valor almacenado en el [LookupResult](#).

4.37.1 Typedef Documentation

4.37.1.1 LookupResult

```
typedef struct LookupResult LookupResult
```

4.37.2 Enumeration Type Documentation

4.37.2.1 Status

```
enum Status
```

Enumerator

OK	
ERROR	
MISS	

Definition at line 4 of file [results.h](#).

```
00004      {
00005          OK,
00006          ERROR,
00007          MISS
00008      } Status;
```

4.37.3 Function Documentation

4.37.3.1 create_error_lookup_result()

`LookupResult create_error_lookup_result () [inline]`

Devuelve un `LookupResult` con valor 0 y status ERROR.

Definition at line 12 of file `results.c`.

```
00012 {
00013     return create_lookup_result(NULL, 0, ERROR);
00014 }
```

4.37.3.2 create_miss_lookup_result()

`LookupResult create_miss_lookup_result () [inline]`

Devuelve un `LookupResult` con valor 0 y status MISS.

Definition at line 16 of file `results.c`.

```
00016 {
00017     return create_lookup_result(NULL, 0, MISS);
00018 }
```

4.37.3.3 create_ok_lookup_result()

`LookupResult create_ok_lookup_result (`
 `void * ptr,`
 `size_t size) [inline]`

Devuelve un `LookupResult` con el valor objetivo y status OK.

Definition at line 8 of file `results.c`.

```
00008 {
00009     return create_lookup_result(ptr, size, OK);
00010 }
```

4.37.3.4 lookup_result_get_size()

`size_t lookup_result_get_size (`
 `LookupResult lr)`

Devuelve el tamaño del valor almacenado en el `LookupResult`.

Definition at line 36 of file `results.c`.

```
00036 {
00037     return lr.size;
00038 }
```

4.37.3.5 lookup_result_get_value()

`void * lookup_result_get_value (`
 `LookupResult lr)`

Devuelve el valor almacenado en el `LookupResult`.

Definition at line 32 of file `results.c`.

```
00032 {
00033     return lr.ptr;
00034 }
```

4.37.3.6 lookup_result_is_error()

```
int lookup_result_is_error (
    LookupResult lr ) [inline]
```

Devuelve 1 si el status del [LookupResult](#) objetivo es ERROR, o 0 en caso contrario.

Definition at line 20 of file [results.c](#).

```
00020                                     {
00021     return lr.status == ERROR;
00022 }
```

4.37.3.7 lookup_result_is_miss()

```
int lookup_result_is_miss (
    LookupResult lr ) [inline]
```

Devuelve 1 si el status del [LookupResult](#) objetivo es MISS, o 0 en caso contrario.

Definition at line 28 of file [results.c](#).

```
00028                                     {
00029     return lr.status == MISS;
00030 }
```

4.37.3.8 lookup_result_is_ok()

```
int lookup_result_is_ok (
    LookupResult lr ) [inline]
```

Devuelve 1 si el status del [LookupResult](#) objetivo es OK, o 0 en caso contrario.

Definition at line 24 of file [results.c](#).

```
00024                                     {
00025     return lr.status == OK;
00026 }
```

4.38 results.h

[Go to the documentation of this file.](#)

```
00001 #ifndef __RESULTS_H__
00002 #define __RESULTS_H__
00003
00004 typedef enum {
00005     OK,
00006     ERROR,
00007     MISS
00008 } Status;
00009
00010 typedef struct LookupResult {
00011     void* ptr;
00012     size_t size;
00013     Status status;
00014 } LookupResult;
00015
00016
00017
00018
00019 /// @brief Devuelve un LookupResult con el valor objetivo y status OK.
00020 LookupResult create_ok_lookup_result(void* ptr, size_t size);
00021
00022 /// @brief Devuelve un LookupResult con valor 0 y status ERROR.
00023 LookupResult create_error_lookup_result();
00024
```



```

00025 /// @brief Devuelve un LookupResult con valor 0 y status MISS.
00026 LookupResult create_miss_lookup_result();
00027
00028 /// @brief Devuelve 1 si el status del LookupResult objetivo es ERROR, o 0 en caso contrario.
00029 int lookup_result_is_error(LookupResult lr);
00030
00031 /// @brief Devuelve 1 si el status del LookupResult objetivo es OK, o 0 en caso contrario.
00032 int lookup_result_is_ok(LookupResult lr);
00033
00034 /// @brief Devuelve 1 si el status del LookupResult objetivo es MISS, o 0 en caso contrario.
00035 int lookup_result_is_miss(LookupResult lr);
00036
00037 /// @brief Devuelve el valor almacenado en el LookupResult.
00038 void* lookup_result_get_value(LookupResult lr);
00039
00040 /// @brief Devuelve el tamaño del valor almacenado en el LookupResult.
00041 size_t lookup_result_get_size(LookupResult lr);
00042
00043 #endif // __RESULTS_H__

```

4.39 lru/lru.c File Reference

```

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "lru.h"
#include "lrunode.h"

```

Classes

- struct [LRUQueue](#)

Functions

- int [lru_queue_lock](#) ([LRUQueue](#) q)
Lockea la [LRUQueue](#) objetivo.
- int [lru_queue_unlock](#) ([LRUQueue](#) q)
Libera el lock de la [LRUQueue](#) objetivo.
- [LRUQueue lru_queue_create](#) ()
Crea una LRU vacia.
- [LRUNode lru_queue_set_most_recent](#) ([LRUNode](#) node, [LRUQueue](#) q)
*Establece al nodo objetivo como el mas reciente de la cola, asumiendo que puede ya haber formado parte de la cola. Esta funcion es *thread-safe*.*
- int [lru_queue_delete_node](#) ([LRUNode](#) node, [LRUQueue](#) q)
Elimina el nodo objetivo de la cola LRU, liberando su memoria. No modifica la memoria asignada a su [HashNode](#) asociado.
- void [lru_queue_node_clean](#) ([LRUNode](#) node, [LRUQueue](#) q)
Limpia un nodo de la [LRUQueue](#). Es decir, lo desconecta de sus vecinos, si es que tiene.
- [LRUNode lru_queue_get_least_recent](#) ([LRUQueue](#) q)
Obtiene el puntero al nodo menos recientemente utilizado de la [LRUQueue](#) objetivo.
- int [lru_queue_destroy](#) ([LRUQueue](#) q)
*Destruye la cola LRU objetivo, liberando los recursos asignados sin liberar la memoria asociada a los nodos de la tabla hash. Esta operacion es *thread-safe*, no debe poseerse el lock al invocarla.*

4.39.1 Function Documentation

4.39.1.1 lru_queue_create()

```
LRUQueue lru_queue_create ( )
```

Crea una LRU vacia.

Returns

La [LRUQueue](#) generada.

Definition at line 23 of file [lru.c](#).

```
00023 {
00024
00025     pthread_mutex_t* lock = malloc(sizeof(pthread_mutex_t));
00026     if (lock == NULL)
00027         return NULL;
00028
00029     if (pthread_mutex_init(lock, NULL) != 0) {
00030         free(lock);
00031         return NULL;
00032     }
00033
00034     LRUQueue queue = malloc(sizeof(struct LRUQueue));
00035     if (queue == NULL) {
00036         free(lock);
00037         return NULL;
00038     }
00039
00040     memset(queue, 0, sizeof(struct LRUQueue));
00041     queue->lock = lock;
00042
00043     return queue;
00044
00045 }
```

4.39.1.2 lru_queue_delete_node()

```
int lru_queue_delete_node (
    LRUNode node,
    LRUQueue q )
```

Elimina el nodo objetivo de la cola LRU, liberando su memoria. No modifica la memoria asignada a su [HashNode](#) asociado.

Esta funcion es thread-safe. No debe poseerse el lock de la [LRUQueue](#) al invocarla.

Parameters

<i>node</i>	El nodo a eliminar.
<i>q</i>	La cola LRU a la que pertenece.

Returns

0 en caso de exito, -1 si se produjo un error.

Definition at line 80 of file [lru.c](#).

```
00080 {
00081
```

```

00082     if (q == NULL || node == NULL)
00083         return 1;
00084
00085     lru_queue_lock(q);
00086
00087     lru_queue_node_clean(node, q);
00088
00089     lru_queue_unlock(q);
00090
00091     lrunode_destroy(node);
00092
00093     return 0;
00094 }
00095

```

4.39.1.3 lru_queue_destroy()

```

int lru_queue_destroy (
    LRUQueue q )

```

Destruye la cola LRU objetivo, liberando los recursos asignados sin liberar la memoria asociada a los nodos de la tabla hash. Esta operacion es `thread-safe`, no debe poseerse el lock al invocarla.

Parameters

<i>q</i>	La cola LRU a destruir.
----------	-------------------------

Returns

0 si es exitoso, -1 en caso de error.

Definition at line 123 of file `lru.c`.

```

00123                                     {
00124
00125     // Solo destruye la memoria que es propia de la LRU. No se mete con la Hash.
00126     if (q == NULL)
00127         return -1;
00128
00129     lru_queue_lock(q);
00130
00131     LRUNode tmp = q->least_recent;
00132     LRUNode next;
00133
00134     while (tmp) {
00135         next = lrunode_get_next(tmp);
00136         lrunode_destroy(tmp);
00137         tmp = next;
00138     }
00139
00140     lru_queue_unlock(q);
00141
00142     // Destruimos el mutex
00143     pthread_mutex_destroy(q->lock);
00144     free(q->lock);
00145
00146     free(q);
00147
00148     return 0;
00149 }
00150

```

4.39.1.4 lru_queue_get_least_recent()

```

LRUNode lru_queue_get_least_recent (
    LRUQueue q )

```

Obtiene el puntero al nodo menos recientemente utilizado de la `LRUQueue` objetivo.

IMPORTANTE Debe contarse con el lock de la `LRUQueue` al invocarse a esta funcion.

Parameters

<i>q</i>	La LRUQueue objetivo.
----------	---------------------------------------

Returns

Un puntero al nodo menos utilizado, que es NULL en caso de que la cola este vacia.

Definition at line 117 of file [lru.c](#).

```
00117                                     {
00118     PRINT("La queue es NULL? %s.", q == NULL ? "Si" : "No");
00119     return (q == NULL) ? NULL : q->least_recent;
00120 }
```

4.39.1.5 lru_queue_lock()

```
int lru_queue_lock (
    LRUQueue q ) [inline]
```

Lockea la [LRUQueue](#) objetivo.

4.39.1.6 Funciones auxiliares

Definition at line 159 of file [lru.c](#).

```
00159                                     {
00160     return (q == NULL) ? -1 : pthread_mutex_lock(q->lock);
00161 }
```

4.39.1.7 lru_queue_node_clean()

```
void lru_queue_node_clean (
    LRUNode node,
    LRUQueue q )
```

Limpia un nodo de la [LRUQueue](#). Es decir, lo desconecta de sus vecinos, si es que tiene.

IMPORTANTE Esta funcion NO es thread-safe. Debe de lockearse la [LRUQueue](#) previo a ser invocada.

Parameters

<i>node</i>	El nodo a limpiar.
<i>q</i>	La cola LRU a la que pertenece.

Definition at line 97 of file [lru.c](#).

```
00097                                     {
00098
00099     LRUNode prev = lrunode_get_prev(node);
00100     LRUNode next = lrunode_get_next(node);
00101
00102     lrunode_set_next(prev, next);
00103     lrunode_set_prev(next, prev);
00104
00105     LRUNode lr = q->least_recent;
00106     LRUNode mr = q->most_recent;
00107
00108     if (lr == node)
00109         q->least_recent = next;
00110
00111     if (mr == node)
00112         q->most_recent = prev;
00113
00114 }
```

4.39.1.8 lru_queue_set_most_recent()

```
LRUNode lru_queue_set_most_recent (
    LRUNode node,
    LRUQueue q )
```

Establece al nodo objetivo como el mas reciente de la cola, asumiendo que puede ya haber formado parte de la cola. Esta funcion es `thread-safe`.

Parameters

<i>node</i>	El nodo objetivo.
<i>q</i>	La cola LRU.

Returns

El puntero al nodo establecido.

I. El previo lru de node pasa a ser el mas reciente de q. II. Si el mas reciente es no nulo, su siguiente pasa a ser node. III. El mas reciente de la cola pasa a ser node. IV. El siguiente de node es siempre NULL.

Definition at line 47 of file `lru.c`.

```
00047                                     {
00048
00049     if (node == NULL || q == NULL)
00050         return NULL;
00051
00052     lru_queue_lock(q);
00053
00054     // Si ya formaba parte de la cola, lo desconectamos.
00055     lru_queue_node_clean(node, q);
00056
00057     /** I.     El previo lru de node pasa a ser el mas reciente de q.
00058      * II.     Si el mas reciente es no nulo, su siguiente pasa a ser node.
00059      * III.    El mas reciente de la cola pasa a ser node.
00060      * IV.     El siguiente de node es siempre NULL.
00061      */
00062     lrunode_set_prev(node, q->most_recent);
00063     lrunode_set_next(q->most_recent, node);
00064
00065     q->most_recent = node;
00066
00067     // Si es el primer nodo a insertar, tambien es el menos reciente.
00068     if (q->least_recent == NULL)
00069         q->least_recent = node;
00070
00071     lrunode_set_next(node, NULL);
00072
00073     lru_queue_unlock(q);
00074
00075     return node;
00076
00077 }
```

4.39.1.9 lru_queue_unlock()

```
int lru_queue_unlock (
    LRUQueue q ) [inline]
```

Libera el lock de la `LRUQueue` objetivo.

Parameters

<i>q</i>	La <code>LRUQueue</code> cuyo lock queremos liberar.
----------	--

Returns

0 en caso exitoso, -1 si el puntero es NULL, un codigo de error en otro caso.

Definition at line 164 of file `lru.c`.

```
00164                                     {
00165     return (q == NULL) ? -1 : pthread_mutex_unlock(q->lock);
00166 }
```

4.40 lru.c

[Go to the documentation of this file.](#)

```

00001 #include <pthread.h>
00002 #include <stdio.h>
00003 #include <stdlib.h>
00004 #include <string.h>
00005 #include "lru.h"
00006 #include "lrunode.h"
00007
00008
00009 struct LRUQueue {
00010     LRUNode most_recent;
00011     LRUNode least_recent;
00012
00013     pthread_mutex_t* lock;
00014 };
00015
00016 inline int lru_queue_lock(LRUQueue q);
00017 inline int lru_queue_unlock(LRUQueue q);
00018
00019 LRUQueue lru_queue_create() {
00020     pthread_mutex_t* lock = malloc(sizeof(pthread_mutex_t));
00021     if (lock == NULL)
00022         return NULL;
00023     if (pthread_mutex_init(lock, NULL) != 0) {
00024         free(lock);
00025         return NULL;
00026     }
00027     LRUQueue queue = malloc(sizeof(struct LRUQueue));
00028     if (queue == NULL) {
00029         free(lock);
00030         return NULL;
00031     }
00032     memset(queue, 0, sizeof(struct LRUQueue));
00033     queue->lock = lock;
00034     return queue;
00035 }
00036
00037 LRUNode lru_queue_set_most_recent(LRUNode node, LRUQueue q) {
00038     if (node == NULL || q == NULL)
00039         return NULL;
00040     lru_queue_lock(q);
00041     // Si ya formaba parte de la cola, lo desconectamos.
00042     lru_queue_node_clean(node, q);
00043     /** I. El previo lru de node pasa a ser el mas reciente de q.
00044      * II. Si el mas reciente es no nulo, su siguiente pasa a ser node.
00045      * III. El mas reciente de la cola pasa a ser node.
00046      * IV. El siguiente de node es siempre NULL.
00047      */
00048     lrunode_set_prev(node, q->most_recent);
00049     lrunode_set_next(q->most_recent, node);
00050     q->most_recent = node;
00051     // Si es el primer nodo a insertar, tambien es el menos reciente.
00052     if (q->least_recent == NULL)
00053         q->least_recent = node;
00054     lrunode_set_next(node, NULL);
00055     lru_queue_unlock(q);
00056     return node;
00057 }
00058
00059 int lru_queue_delete_node(LRUNode node, LRUQueue q) {
00060     if (q == NULL || node == NULL)
00061         return 1;

```

```

00084
00085     lru_queue_lock(q);
00086
00087     lru_queue_node_clean(node, q);
00088
00089     lru_queue_unlock(q);
00090
00091     lrunode_destroy(node);
00092
00093     return 0;
00094 }
00095
00096 void lru_queue_node_clean(LRUNode node, LRUQueue q) {
00097     LRUNode prev = lrunode_get_prev(node);
00098     LRUNode next = lrunode_get_next(node);
00099
00100     lrunode_set_next(prev, next);
00101     lrunode_set_prev(next, prev);
00102
00103     LRUNode lr = q->least_recent;
00104     LRUNode mr = q->most_recent;
00105
00106     if (lr == node)
00107         q->least_recent = next;
00108     if (mr == node)
00109         q->most_recent = prev;
00110 }
00111
00112 LRUNode lru_queue_get_least_recent(LRUQueue q) {
00113     PRINT("La queue es NULL? %s.", q == NULL ? "Si" : "No");
00114     return (q == NULL) ? NULL : q->least_recent;
00115 }
00116
00117 int lru_queue_destroy(LRUQueue q) {
00118     // Solo destruye la memoria que es propia de la LRU. No se mete con la Hash.
00119     if (q == NULL)
00120         return -1;
00121
00122     lru_queue_lock(q);
00123
00124     LRUNode tmp = q->least_recent;
00125     LRUNode next;
00126
00127     while (tmp) {
00128         next = lrunode_get_next(tmp);
00129         lrunode_destroy(tmp);
00130         tmp = next;
00131     }
00132
00133     lru_queue_unlock(q);
00134
00135     // Destruimos el mutex
00136     pthread_mutex_destroy(q->lock);
00137     free(q->lock);
00138
00139     free(q);
00140
00141     return 0;
00142 }
00143
00144 /** -----
00145  *           Funciones auxiliares
00146  * -----
00147  */
00148
00149 inline int lru_queue_lock(LRUQueue q) {
00150     return (q == NULL) ? -1 : pthread_mutex_lock(q->lock);
00151 }
00152
00153 inline int lru_queue_unlock(LRUQueue q) {
00154     return (q == NULL) ? -1 : pthread_mutex_unlock(q->lock);
00155 }
00156
00157
00158
00159
00160
00161
00162
00163
00164
00165
00166
00167
00168
00169
00170

```

4.41 Iru/Iru.h File Reference

```
#include "../src/cache/cache.h"
#include "../dynalloc/dynalloc.h"
```

Typedefs

- typedef struct [HashNode](#) * [HashNode](#)
- typedef struct [LRUNode](#) * [LRUNode](#)
- typedef struct [LRUQueue](#) * [LRUQueue](#)

Functions

- [LRUQueue](#) [lru_queue_create](#) ()
Crea una LRU vacia.
- int [lru_queue_lock](#) ([LRUQueue](#) q)
Lockea la [LRUQueue](#) objetivo.
- int [lru_queue_unlock](#) ([LRUQueue](#) q)
Libera el lock de la [LRUQueue](#) objetivo.
- [LRUNode](#) [lru_queue_set_most_recent](#) ([LRUNode](#) node, [LRUQueue](#) q)
*Establece al nodo objetivo como el mas reciente de la cola, asumiendo que puede ya haber formado parte de la cola. Esta funcion es *thread-safe*.*
- [LRUNode](#) [lru_queue_get_least_recent](#) ([LRUQueue](#) q)
Obtiene el puntero al nodo menos recientemente utilizado de la [LRUQueue](#) objetivo.
- int [lru_queue_destroy](#) ([LRUQueue](#) q)
*Destruye la cola LRU objetivo, liberando los recursos asignados sin liberar la memoria asociada a los nodos de la tabla hash. Esta operacion es *thread-safe*, no debe poseerse el lock al invocarla.*
- int [lru_queue_delete_node](#) ([LRUNode](#) node, [LRUQueue](#) q)
Elimina el nodo objetivo de la cola LRU, liberando su memoria. No modifica la memoria asignada a su [HashNode](#) asociado.
- void [lru_queue_node_clean](#) ([LRUNode](#) node, [LRUQueue](#) q)
Limpia un nodo de la [LRUQueue](#). Es decir, lo desconecta de sus vecinos, si es que tiene.

4.41.1 Typedef Documentation

4.41.1.1 HashNode

```
typedef struct HashNode* HashNode
```

Definition at line 8 of file [lru.h](#).

4.41.1.2 LRUNode

```
typedef struct LRUNode* LRUNode
```

Definition at line 9 of file [lru.h](#).

4.41.1.3 LRUQueue

```
typedef struct LRUQueue* LRUQueue
```

Definition at line 12 of file [lru.h](#).

4.41.2 Function Documentation

4.41.2.1 lru_queue_create()

```
LRUQueue lru\_queue\_create ( )
```

Crea una LRU vacia.

Returns

La [LRUQueue](#) generada.

Definition at line 23 of file [lru.c](#).

```

00023         {
00024
00025     pthread_mutex_t* lock = malloc(sizeof(pthread_mutex_t));
00026     if (lock == NULL)
00027         return NULL;
00028
00029     if (pthread_mutex_init(lock, NULL) != 0) {
00030         free(lock);
00031         return NULL;
00032     }
00033
00034     LRUQueue queue = malloc(sizeof(struct LRUQueue));
00035     if (queue == NULL) {
00036         free(lock);
00037         return NULL;
00038     }
00039
00040     memset(queue, 0, sizeof(struct LRUQueue));
00041     queue->lock = lock;
00042
00043     return queue;
00044 }
00045 }
```

4.41.2.2 lru_queue_delete_node()

```

int lru_queue_delete_node (
    LRUNode node,
    LRUQueue q )
```

Elimina el nodo objetivo de la cola LRU, liberando su memoria. No modifica la memoria asignada a su [HashNode](#) asociado.

Esta funcion es thread-safe. No debe poseerse el lock de la [LRUQueue](#) al invocarla.

Parameters

<i>node</i>	El nodo a eliminar.
<i>q</i>	La cola LRU a la que pertenece.

Returns

0 en caso de exito, -1 si se produjo un error.

Definition at line 80 of file [lru.c](#).

```

00080         {
00081
00082     if (q == NULL || node == NULL)
00083         return 1;
00084
00085     lru_queue_lock(q);
00086
00087     lru_queue_node_clean(node, q);
00088
00089     lru_queue_unlock(q);
00090
00091     lrunode_destroy(node);
00092
00093     return 0;
00094 }
00095 }
```

4.41.2.3 lru_queue_destroy()

```

int lru_queue_destroy (
    LRUQueue q )
```

Destruye la cola LRU objetivo, liberando los recursos asignados sin liberar la memoria asociada a los nodos de la tabla hash. Esta operacion es thread-safe, no debe poseerse el lock al invocarla.

Parameters

<i>q</i>	La cola LRU a destruir.
----------	-------------------------

Returns

0 si es exitoso, -1 en caso de error.

Definition at line 123 of file [lru.c](#).

```

00123                                     {
00124
00125     // Solo destruye la memoria que es propia de la LRU. No se mete con la Hash.
00126     if (q == NULL)
00127         return -1;
00128
00129     lru_queue_lock(q);
00130
00131     LRUNode tmp = q->least_recent;
00132     LRUNode next;
00133
00134     while (tmp) {
00135         next = lrunode_get_next(tmp);
00136         lrunode_destroy(tmp);
00137         tmp = next;
00138     }
00139
00140     lru_queue_unlock(q);
00141
00142     // Destruimos el mutex
00143     pthread_mutex_destroy(q->lock);
00144     free(q->lock);
00145
00146     free(q);
00147
00148     return 0;
00149
00150 }
```

4.41.2.4 lru_queue_get_least_recent()

```

LRUNode lru_queue_get_least_recent (
    LRUQueue q )
```

Obtiene el puntero al nodo menos recientemente utilizado de la [LRUQueue](#) objetivo.

IMPORTANTE Debe contarse con el lock de la [LRUQueue](#) al invocarse a esta funcion.

Parameters

<i>q</i>	La LRUQueue objetivo.
----------	---------------------------------------

Returns

Un puntero al nodo menos utilizado, que es NULL en caso de que la cola este vacia.

Definition at line 117 of file [lru.c](#).

```

00117                                     {
00118     PRINT("La queue es NULL? %s.", q == NULL ? "Si" : "No");
00119     return (q == NULL) ? NULL : q->least_recent;
00120 }
```

4.41.2.5 lru_queue_lock()

```

int lru_queue_lock (
    LRUQueue q ) [inline]
```

Lockea la [LRUQueue](#) objetivo.

Parameters

<i>q</i>	La LRUQueue a lockear.
----------	--

Returns

0 en caso exitoso, -1 si el puntero es NULL, un codigo de error en otro caso.

4.41.2.6 Funciones auxiliares

Definition at line 159 of file [lru.c](#).

```
00159      {
00160  return (q == NULL) ? -1 : pthread_mutex_lock(q->lock);
00161 }
```

4.41.2.7 lru_queue_node_clean()

```
void lru_queue_node_clean (
    LRUNode node,
    LRUQueue q )
```

Limpia un nodo de la [LRUQueue](#). Es decir, lo desconecta de sus vecinos, si es que tiene.

IMPORTANTE Esta funcion NO es thread-safe. Debe de lockearse la [LRUQueue](#) previo a ser invocada.

Parameters

<i>node</i>	El nodo a limpiar.
<i>q</i>	La cola LRU a la que pertenece.

Definition at line 97 of file [lru.c](#).

```
00097      {
00098
00099  LRUNode prev = lrunode_get_prev(node);
00100  LRUNode next = lrunode_get_next(node);
00101
00102  lrunode_set_next(prev, next);
00103  lrunode_set_prev(next, prev);
00104
00105  LRUNode lr = q->least_recent;
00106  LRUNode mr = q->most_recent;
00107
00108  if (lr == node)
00109      q->least_recent = next;
00110
00111  if (mr == node)
00112      q->most_recent = prev;
00113
00114 }
```

4.41.2.8 lru_queue_set_most_recent()

```
LRUNode lru_queue_set_most_recent (
    LRUNode node,
    LRUQueue q )
```

Establece al nodo objetivo como el mas reciente de la cola, asumiendo que puede ya haber formado parte de la cola. Esta funcion es thread-safe.

Parameters

<i>node</i>	El nodo objetivo.
<i>q</i>	La cola LRU.

Returns

El puntero al nodo establecido.

I. El previo lru de node pasa a ser el mas reciente de q. II. Si el mas reciente es no nulo, su siguiente pasa a ser node. III. El mas reciente de la cola pasa a ser node. IV. El siguiente de node es siempre NULL.

Definition at line 47 of file [lru.c](#).

```
00047      {
00048
```

```

00049  if (node == NULL || q == NULL)
00050      return NULL;
00051
00052  lru_queue_lock(q);
00053
00054  // Si ya formaba parte de la cola, lo desconectamos.
00055  lru_queue_node_clean(node, q);
00056
00057  /** I.    El previo lru de node pasa a ser el mas reciente de q.
00058   * II.    Si el mas reciente es no nulo, su siguiente pasa a ser node.
00059   * III.   El mas reciente de la cola pasa a ser node.
00060   * IV.    El siguiente de node es siempre NULL.
00061   */
00062  lrunode_set_prev(node, q->most_recent);
00063  lrunode_set_next(q->most_recent, node);
00064
00065  q->most_recent = node;
00066
00067  // Si es el primer nodo a insertar, tambien es el menos reciente.
00068  if (q->least_recent == NULL)
00069      q->least_recent = node;
00070
00071  lrunode_set_next(node, NULL);
00072
00073  lru_queue_unlock(q);
00074
00075  return node;
00076
00077 }

```

4.41.2.9 lru_queue_unlock()

```

int lru_queue_unlock (
    LRUQueue q ) [inline]

```

Libera el lock de la [LRUQueue](#) objetivo.

Parameters

<i>q</i>	La LRUQueue cuyo lock queremos liberar.
----------	---

Returns

0 en caso exitoso, -1 si el puntero es NULL, un codigo de error en otro caso.

Definition at line [164](#) of file [lru.c](#).

```

00164  {
00165  return (q == NULL) ? -1 : pthread_mutex_unlock(q->lock);
00166  }

```

4.42 lru.h

[Go to the documentation of this file.](#)

```

00001 #ifndef __LRU_H__
00002 #define __LRU_H__
00003
00004 #include "../src/cache/cache.h"
00005 #include "../dynalloc/dynalloc.h"
00006
00007 // Forward-declarations para no incluir sus header files
00008 typedef struct HashNode* HashNode;
00009 typedef struct LRUNode* LRUNode;
00010
00011 // Estructuras opacas
00012 typedef struct LRUQueue* LRUQueue;
00013
00014 /**
00015  * @brief Crea una LRU vacia.
00016  *
00017  * @return La LRUQueue generada.
00018  *
00019  */
00020 LRUQueue lru_queue_create();
00021
00022 /**
00023  * @brief Lockea la LRUQueue objetivo.

```

```

00025 * @param q La LRUQueue a lockear.
00026 *
00027 * @return 0 en caso exitoso, -1 si el puntero es NULL, un codigo de error en otro caso.
00028 */
00029 int lru_queue_lock(LRUQueue q);
00030
00031
00032 /**
00033 * @brief Libera el lock de la LRUQueue objetivo.
00034 * @param q La LRUQueue cuyo lock queremos liberar.
00035 *
00036 * @return 0 en caso exitoso, -1 si el puntero es NULL, un codigo de error en otro caso.
00037 */
00038 int lru_queue_unlock(LRUQueue q);
00039
00040
00041 /**
00042 * @brief Establece al nodo objetivo como el mas reciente de la cola, asumiendo que puede ya haber
    formado parte de la cola. Esta funcion es `thread-safe`.
00043 *
00044 * @param node El nodo objetivo.
00045 * @param q La cola LRU.
00046 *
00047 * @return El puntero al nodo establecido.
00048 *
00049 */
00050 LRUNode lru_queue_set_most_recent(LRUNode node, LRUQueue q);
00051
00052 /**
00053 * @brief Obtiene el puntero al nodo menos recientemente utilizado de la LRUQueue objetivo.
00054 *
00055 * `IMPORTANTE` Debe contarse con el lock de la LRUQueue al invocarse a esta funcion.
00056 *
00057 * @param q La LRUQueue objetivo.
00058 * @return Un puntero al nodo menos utilizado, que es NULL en caso de que la cola este vacia.
00059 *
00060 */
00061 LRUNode lru_queue_get_least_recent(LRUQueue q);
00062
00063 /**
00064 * @brief Destruye la cola LRU objetivo, liberando los recursos asignados sin liberar la memoria
    asociada a los nodos de la tabla hash. Esta operacion es `thread-safe`, no debe poseerse el lock al
    invocarla.
00065 *
00066 * @param q La cola LRU a destruir.
00067 * @return 0 si es exitoso, -1 en caso de error.
00068 */
00069 int lru_queue_destroy(LRUQueue q);
00070
00071 /**
00072 * @brief Elimina el nodo objetivo de la cola LRU, liberando su memoria. No modifica la memoria
    asignada a su HashNode asociado.
00073 *
00074 * Esta funcion es thread-safe. No debe poseerse el lock de la LRUQueue al invocarla.
00075 *
00076 * @param node El nodo a eliminar.
00077 * @param q La cola LRU a la que pertenece.
00078 *
00079 * @return 0 en caso de exito, -1 si se produjo un error.
00080 */
00081 int lru_queue_delete_node(LRUNode node, LRUQueue q);
00082
00083 /**
00084 * @brief Limpia un nodo de la LRUQueue. Es decir, lo desconecta de sus vecinos, si es que tiene.
00085 *
00086 * `IMPORTANTE` Esta funcion NO es thread-safe. Debe de lockearse la LRUQueue previo a ser invocada.
00087 *
00088 * @param node El nodo a limpiar.
00089 * @param q La cola LRU a la que pertenece.
00090 */
00091 void lru_queue_node_clean(LRUNode node, LRUQueue q);
00092
00093 #endif // __LRU_H__

```

4.43 lru/lrunode.c File Reference

```

#include <string.h>
#include <stdio.h>
#include "lrunode.h"
#include "../dynalloc/dynalloc.h"

```

Classes

- struct [LRUNode](#)

Macros

- #define [PRINT](#)(fmt, ...) printf("[%s] " fmt "\n", __func__, ##__VA_ARGS__)

Functions

- [LRUNode lrunode_create](#) ([Cache](#) cache)
Crea un nuevo [LRUNode](#) vacío.
- int [lru_node_is_clean](#) ([LRUNode](#) node)
Devuelve un booleano representando si el nodo LRU esta 'limpio'. Diremos que esta limpio si no esta conectado a ningun nodo LRU.
- void [lrunode_destroy](#) ([LRUNode](#) node)
Destruye el nodo objetivo, liberando la memoria asociada. Se asume que el nodo ya esta 'limpio'.
- [LRUNode lrunode_get_prev](#) ([LRUNode](#) node)
Obtiene el nodo previo.
- void [lrunode_set_prev](#) ([LRUNode](#) node, [LRUNode](#) prev)
Establece el nodo previo.
- [LRUNode lrunode_get_next](#) ([LRUNode](#) node)
Obtiene el nodo siguiente.
- void [lrunode_set_next](#) ([LRUNode](#) node, [LRUNode](#) next)
Establece el nodo siguiente.
- [HashNode lrunode_get_hash_node](#) ([LRUNode](#) node)
Obtiene el nodo hash asociado.
- void [lrunode_set_hash_node](#) ([LRUNode](#) node, [HashNode](#) hash_node)
Establece el nodo hash asociado.
- void [lrunode_set_bucket_number](#) ([LRUNode](#) node, unsigned int bucket_number)
Establece el numero de bucket del nodo objetivo.
- unsigned int [lrunode_get_bucket_number](#) ([LRUNode](#) node)
*Obtiene el numero de bucket del hashnode asociado a *node*.*

4.43.1 Macro Definition Documentation

4.43.1.1 PRINT

```
#define PRINT(  
    fmt,  
    ... ) printf("[%s] " fmt "\n", __func__, ##__VA_ARGS__)
```

Definition at line 6 of file [lrunode.c](#).

4.43.2 Function Documentation

4.43.2.1 lru_node_is_clean()

```
int lru_node_is_clean (  
    LRUNode node )
```

Devuelve un booleano representando si el nodo LRU esta 'limpio'. Diremos que esta limpio si no esta conectado a ningun nodo LRU.

Parameters

<i>node</i>	El nodo LRU objetivo.
-------------	-----------------------

Returns

1 si esta limpio, 0 si no.

Definition at line 30 of file [lrunode.c](#).

```
00030                                     {
00031     if (node == NULL) return 1;
00032     return node->prev == NULL &&
00033            node->next == NULL;
00034 }
```

4.43.2.2 lrunode_create()

```
LRUNode lrunode_create (
    Cache cache )
```

Crea un nuevo [LRUNode](#) vacio.

Parameters

Cache	La cache a la que pertenece la LRUQueue donde se insertara el nodo.
-----------------------	---

Returns

Un puntero al [LRUNode](#) nuevo.

Definition at line 18 of file [lrunode.c](#).

```
00018                                     {
00019
00020     LRUNode node = dynalloc(sizeof(struct LRUNode), cache);
00021     if (node == NULL)
00022         return NULL;
00023
00024     memset(node, 0, sizeof(struct LRUNode));
00025
00026     return node;
00027
00028 }
```

4.43.2.3 lrunode_destroy()

```
void lrunode_destroy (
    LRUNode node )
```

Destruye el nodo objetivo, liberando la memoria asociada. Se asume que el nodo ya esta 'limpio'.

Parameters

<i>node</i>	El nodo a destruir.
-------------	---------------------

Definition at line 36 of file [lrunode.c](#).

```
00036                                     {
00037     if (node != NULL)
00038         free(node);
00039 }
```

4.43.2.4 lrunode_get_bucket_number()

```
unsigned int lrunode_get_bucket_number (
    LRUNode node )
```

Obtiene el numero de bucket del hashnode asociado a node.

Parameters

<i>node</i>	Puntero al nodo LRU.
-------------	----------------------

Returns

El nmero de bucket del hash node asociado a `node`.

Definition at line 85 of file [lrunode.c](#).

```
00085                                     {
00086
00087     // ! BORRAR
00088     if (node)
00089         PRINT("Bucket number calculado para el node a desalojar: %u", node->bucket_number);
00090
00091     return node == NULL? 0 : node->bucket_number;
00092 }
```

4.43.2.5 lrunode_get_hash_node()

```
HashNode lrunode_get_hash_node (
    LRUNode node )
```

Obtiene el nodo hash asociado.

Parameters

<code>node</code>	Puntero al nodo LRU.
-------------------	----------------------

Returns

Puntero al nodo hash.

Definition at line 67 of file [lrunode.c](#).

```
00067                                     {
00068     if (node == NULL) {
00069         return NULL;
00070     }
00071     return node->hash_node;
00072 }
```

4.43.2.6 lrunode_get_next()

```
LRUNode lrunode_get_next (
    LRUNode node )
```

Obtiene el nodo siguiente.

Parameters

<code>node</code>	Puntero al nodo LRU.
-------------------	----------------------

Returns

Puntero al nodo siguiente.

Definition at line 54 of file [lrunode.c](#).

```
00054                                     {
00055     if (node == NULL) {
00056         return NULL;
00057     }
00058     return node->next;
00059 }
```

4.43.2.7 lrunode_get_prev()

```
LRUNode lrunode_get_prev (
    LRUNode node )
```

Obtiene el nodo previo.

Parameters

<code>node</code>	Puntero al nodo LRU.
-------------------	----------------------

Returns

Puntero al nodo previo.

Definition at line 41 of file [lrunode.c](#).

```
00041                                     {
00042     if (node == NULL) {
00043         return NULL;
00044     }
00045     return node->prev;
00046 }
```

4.43.2.8 lrunode_set_bucket_number()

```
void lrunode_set_bucket_number (
    LRUNode node,
    unsigned int bucket_number )
```

Establece el numero de bucket del nodo objetivo.

Parameters

<i>node</i>	Puntero al nodo LRU.
<i>bucket_number</i>	Numero de bucket del hash node asociado a <i>node</i> .

Definition at line 80 of file [lrunode.c](#).

```
00080                                     {
00081     if (node != NULL)
00082         node->bucket_number = bucket_number;
00083 }
```

4.43.2.9 lrunode_set_hash_node()

```
void lrunode_set_hash_node (
    LRUNode node,
    HashNode hash_node )
```

Establece el nodo hash asociado.

Parameters

<i>node</i>	Puntero al nodo LRU.
<i>hash_node</i>	Puntero al nuevo nodo hash.

Definition at line 74 of file [lrunode.c](#).

```
00074                                     {
00075     if (node != NULL) {
00076         node->hash_node = hash_node;
00077     }
00078 }
```

4.43.2.10 lrunode_set_next()

```
void lrunode_set_next (
    LRUNode node,
    LRUNode next )
```

Establece el nodo siguiente.

Parameters

<i>node</i>	Puntero al nodo LRU.
<i>next</i>	Puntero al nuevo nodo siguiente.

Definition at line 61 of file [lrunode.c](#).

```
00061                                     {
00062     if (node != NULL) {
00063         node->next = next;
```

```
00064     }
00065 }
```

4.43.2.11 lrunode_set_prev()

```
void lrunode_set_prev (
    LRUNode node,
    LRUNode prev )
```

Establece el nodo previo.

Parameters

<i>node</i>	Puntero al nodo LRU.
<i>prev</i>	Puntero al nuevo nodo previo.

Definition at line 48 of file [lrunode.c](#).

```
00048
00049     if (node != NULL) {
00050         node->prev = prev;
00051     }
00052 }
```

4.44 lrunode.c

[Go to the documentation of this file.](#)

```
00001 #include <string.h>
00002 #include <stdio.h>
00003 #include "lrunode.h"
00004 #include "../dynalloc/dynalloc.h"
00005
00006 #define PRINT(fmt, ...) printf("[%s] " fmt "\n", __func__, ##__VA_ARGS__)
00007
00008
00009 struct LRUNode {
00010     struct LRUNode* prev;
00011     struct LRUNode* next;
00012
00013     HashNode hash_node;
00014     unsigned int bucket_number;
00015
00016 };
00017
00018 LRUNode lrunode_create(Cache cache) {
00019
00020     LRUNode node = dynalloc(sizeof(struct LRUNode), cache);
00021     if (node == NULL)
00022         return NULL;
00023
00024     memset(node, 0, sizeof(struct LRUNode));
00025
00026     return node;
00027 }
00028
00029
00030 int lru_node_is_clean(LRUNode node) {
00031     if (node == NULL) return 1;
00032     return node->prev == NULL &&
00033         node->next == NULL;
00034 }
00035
00036 void lrunode_destroy(LRUNode node) {
00037     if (node != NULL)
00038         free(node);
00039 }
00040
00041 LRUNode lrunode_get_prev(LRUNode node) {
00042     if (node == NULL) {
00043         return NULL;
00044     }
00045     return node->prev;
00046 }
00047
00048 void lrunode_set_prev(LRUNode node, LRUNode prev) {
00049     if (node != NULL) {
00050         node->prev = prev;
00051     }
00052 }
```

```

00053
00054 LRUNode lrunode_get_next(LRUNode node) {
00055     if (node == NULL) {
00056         return NULL;
00057     }
00058     return node->next;
00059 }
00060
00061 void lrunode_set_next(LRUNode node, LRUNode next) {
00062     if (node != NULL) {
00063         node->next = next;
00064     }
00065 }
00066
00067 HashNode lrunode_get_hash_node(LRUNode node) {
00068     if (node == NULL) {
00069         return NULL;
00070     }
00071     return node->hash_node;
00072 }
00073
00074 void lrunode_set_hash_node(LRUNode node, HashNode hash_node) {
00075     if (node != NULL) {
00076         node->hash_node = hash_node;
00077     }
00078 }
00079
00080 void lrunode_set_bucket_number(LRUNode node, unsigned int bucket_number) {
00081     if (node != NULL) {
00082         node->bucket_number = bucket_number;
00083     }
00084 }
00085 unsigned int lrunode_get_bucket_number(LRUNode node) {
00086
00087     // ! BORRAR
00088     if (node)
00089         PRINT("Bucket number calculado para el node a desalojar: %u", node->bucket_number);
00090
00091     return node == NULL? 0 : node->bucket_number;
00092 }
00093

```

4.45 lru/lrunode.h File Reference

Typedefs

- typedef struct [Cache](#) * [Cache](#)
- typedef struct [LRUNode](#) * [LRUNode](#)
- typedef struct [HashNode](#) * [HashNode](#)

Functions

- [LRUNode lrunode_create](#) ([Cache](#) cache)
Crea un nuevo [LRUNode](#) vacío.
- [int lru_node_is_clean](#) ([LRUNode](#) node)
Devuelve un booleano representando si el nodo LRU esta 'limpio'. Diremos que esta limpio si no esta conectado a ningun nodo LRU.
- [void lrunode_destroy](#) ([LRUNode](#) node)
Destruye el nodo objetivo, liberando la memoria asociada. Se asume que el nodo ya esta 'limpio'.
- [LRUNode lrunode_get_prev](#) ([LRUNode](#) node)
Obtiene el nodo previo.
- [void lrunode_set_prev](#) ([LRUNode](#) node, [LRUNode](#) prev)
Establece el nodo previo.
- [LRUNode lrunode_get_next](#) ([LRUNode](#) node)
Obtiene el nodo siguiente.
- [void lrunode_set_next](#) ([LRUNode](#) node, [LRUNode](#) next)
Establece el nodo siguiente.
- [HashNode lrunode_get_hash_node](#) ([LRUNode](#) node)
Obtiene el nodo hash asociado.

- void `lrunode_set_hash_node` (`LRUNode` node, `HashNode` hash_node)
Establece el nodo hash asociado.
- unsigned int `lrunode_get_bucket_number` (`LRUNode` node)
Obtiene el numero de bucket del hashnode asociado a `node`.
- void `lrunode_set_bucket_number` (`LRUNode` node, unsigned int bucket_number)
Establece el numero de bucket del nodo objetivo.

4.45.1 Typedef Documentation

4.45.1.1 Cache

typedef struct `Cache*` `Cache`
Definition at line 5 of file `lrunode.h`.

4.45.1.2 HashNode

typedef struct `HashNode*` `HashNode`
Definition at line 8 of file `lrunode.h`.

4.45.1.3 LRUNode

typedef struct `LRUNode*` `LRUNode`
Definition at line 7 of file `lrunode.h`.

4.45.2 Function Documentation

4.45.2.1 lru_node_is_clean()

int `lru_node_is_clean` (
 `LRUNode` node)

Devuelve un booleano representando si el nodo LRU esta 'limpio'. Diremos que esta limpio si no esta conectado a ningun nodo LRU.

Parameters

<code>node</code>	El nodo LRU objetivo.
-------------------	-----------------------

Returns

1 si esta limpio, 0 si no.

Definition at line 30 of file `lrunode.c`.

```
00030
00031     if (node == NULL) return 1;
00032     return node->prev == NULL &&
00033           node->next == NULL;
00034 }
```

4.45.2.2 lrunode_create()

`LRUNode` `lrunode_create` (
 `Cache` cache)

Crea un nuevo `LRUNode` vacio.

Parameters

<code>Cache</code>	La cache a la que pertenece la <code>LRUQueue</code> donde se insertara el nodo.
--------------------	--

Returns

Un puntero al [LRUNode](#) nuevo.

Definition at line 18 of file [lrunode.c](#).

```
00018                                     {
00019
00020     LRUNode node = dynalloc(sizeof(struct LRUNode), cache);
00021     if (node == NULL)
00022         return NULL;
00023
00024     memset(node, 0, sizeof(struct LRUNode));
00025
00026     return node;
00027
00028 }
```

4.45.2.3 lrunode_destroy()

```
void lrunode_destroy (
    LRUNode node )
```

Destruye el nodo objetivo, liberando la memoria asociada. Se asume que el nodo ya esta 'limpio'.

Parameters

<i>node</i>	El nodo a destruir.
-------------	---------------------

Definition at line 36 of file [lrunode.c](#).

```
00036                                     {
00037     if (node != NULL)
00038         free(node);
00039 }
```

4.45.2.4 lrunode_get_bucket_number()

```
unsigned int lrunode_get_bucket_number (
    LRUNode node )
```

Obtiene el numero de bucket del hashnode asociado a node.

Parameters

<i>node</i>	Puntero al nodo LRU.
-------------	----------------------

Returns

El nmero de bucket del hash node asociado a node.

Definition at line 85 of file [lrunode.c](#).

```
00085                                     {
00086
00087     // ! BORRAR
00088     if (node)
00089         PRINT("Bucket number calculado para el node a desalojar: %u", node->bucket_number);
00090
00091     return node == NULL? 0 : node->bucket_number;
00092 }
```

4.45.2.5 lrunode_get_hash_node()

```
HashNode lrunode_get_hash_node (
    LRUNode node )
```

Obtiene el nodo hash asociado.

Parameters

<i>node</i>	Puntero al nodo LRU.
-------------	----------------------

Returns

Puntero al nodo hash.

Definition at line 67 of file [lrunode.c](#).

```
00067                                     {
00068     if (node == NULL) {
00069         return NULL;
00070     }
00071     return node->hash_node;
00072 }
```

4.45.2.6 lrunode_get_next()

```
LRUNode lrunode_get_next (
    LRUNode node )
```

Obtiene el nodo siguiente.

Parameters

<i>node</i>	Puntero al nodo LRU.
-------------	----------------------

Returns

Puntero al nodo siguiente.

Definition at line 54 of file [lrunode.c](#).

```
00054                                     {
00055     if (node == NULL) {
00056         return NULL;
00057     }
00058     return node->next;
00059 }
```

4.45.2.7 lrunode_get_prev()

```
LRUNode lrunode_get_prev (
    LRUNode node )
```

Obtiene el nodo previo.

Parameters

<i>node</i>	Puntero al nodo LRU.
-------------	----------------------

Returns

Puntero al nodo previo.

Definition at line 41 of file [lrunode.c](#).

```
00041                                     {
00042     if (node == NULL) {
00043         return NULL;
00044     }
00045     return node->prev;
00046 }
```

4.45.2.8 lrunode_set_bucket_number()

```
void lrunode_set_bucket_number (
    LRUNode node,
    unsigned int bucket_number )
```

Establece el numero de bucket del nodo objetivo.

Parameters

<i>node</i>	Puntero al nodo LRU.
<i>bucket_number</i>	Numero de bucket del hash node asociado a <i>node</i> .

Definition at line 80 of file [lrunode.c](#).

```
00080                                     {
00081     if (node != NULL)
00082         node->bucket_number = bucket_number;
00083 }
```

4.45.2.9 lrunode_set_hash_node()

```
void lrunode_set_hash_node (
    LRUNode node,
    HashNode hash_node )
```

Establece el nodo hash asociado.

Parameters

<i>node</i>	Puntero al nodo LRU.
<i>hash_node</i>	Puntero al nuevo nodo hash.

Definition at line 74 of file [lrunode.c](#).

```
00074                                     {
00075     if (node != NULL) {
00076         node->hash_node = hash_node;
00077     }
00078 }
```

4.45.2.10 lrunode_set_next()

```
void lrunode_set_next (
    LRUNode node,
    LRUNode next )
```

Establece el nodo siguiente.

Parameters

<i>node</i>	Puntero al nodo LRU.
<i>next</i>	Puntero al nuevo nodo siguiente.

Definition at line 61 of file [lrunode.c](#).

```
00061                                     {
00062     if (node != NULL) {
00063         node->next = next;
00064     }
00065 }
```

4.45.2.11 lrunode_set_prev()

```
void lrunode_set_prev (
    LRUNode node,
    LRUNode prev )
```

Establece el nodo previo.

Parameters

<i>node</i>	Puntero al nodo LRU.
<i>prev</i>	Puntero al nuevo nodo previo.

Definition at line 48 of file [lrunode.c](#).

```
00048                                     {
00049     if (node != NULL) {
00050         node->prev = prev;
00051     }
00052 }
```

4.46 lrunode.h

[Go to the documentation of this file.](#)

```

00001 #ifndef __LRU_NODE_H__
00002 #define __LRU_NODE_H__
00003
00004 // Forward declaration
00005 typedef struct Cache* Cache;
00006
00007 typedef struct LRUNode* LRUNode;
00008 typedef struct HashNode* HashNode;
00009
00010
00011 /**
00012  * @brief Crea un nuevo LRUNode vacio.
00013  *
00014  * @param Cache La cache a la que pertenece la LRUQueue donde se insertara el nodo.
00015  * @return Un puntero al LRUNode nuevo.
00016  */
00017 LRUNode lrunode_create(Cache cache);
00018
00019 /**
00020  * @brief Devuelve un booleano representando si el nodo LRU esta 'limpio'. Diremos que esta limpio si
00021  * no esta conectado a ningun nodo LRU.
00022  *
00023  * @param node El nodo LRU objetivo.
00024  * @return 1 si esta limpio, 0 si no.
00025  */
00026 int lru_node_is_clean(LRUNode node);
00027
00028 /**
00029  * @brief Destruye el nodo objetivo, liberando la memoria asociada. Se asume que el nodo ya esta
00030  * 'limpio'.
00031  *
00032  * @param node El nodo a destruir.
00033  */
00034 void lrunode_destroy(LRUNode node);
00035
00036 /**
00037  * @brief Obtiene el nodo previo.
00038  *
00039  * @param node Puntero al nodo LRU.
00040  * @return Puntero al nodo previo.
00041  */
00042 LRUNode lrunode_get_prev(LRUNode node);
00043
00044 /**
00045  * @brief Establece el nodo previo.
00046  *
00047  * @param node Puntero al nodo LRU.
00048  * @param prev Puntero al nuevo nodo previo.
00049  */
00050 void lrunode_set_prev(LRUNode node, LRUNode prev);
00051
00052 /**
00053  * @brief Obtiene el nodo siguiente.
00054  *
00055  * @param node Puntero al nodo LRU.
00056  * @return Puntero al nodo siguiente.
00057  */
00058 LRUNode lrunode_get_next(LRUNode node);
00059
00060 /**
00061  * @brief Establece el nodo siguiente.
00062  *
00063  * @param node Puntero al nodo LRU.
00064  * @param next Puntero al nuevo nodo siguiente.
00065  */
00066 void lrunode_set_next(LRUNode node, LRUNode next);
00067
00068 /**
00069  * @brief Obtiene el nodo hash asociado.
00070  *
00071  * @param node Puntero al nodo LRU.
00072  * @return Puntero al nodo hash.
00073  */
00074 HashNode lrunode_get_hash_node(LRUNode node);
00075
00076 /**
00077  * @brief Establece el nodo hash asociado.
00078  *
00079  * @param node Puntero al nodo LRU.
00080  * @param hash_node Puntero al nuevo nodo hash.
00081  */
00082 void lrunode_set_hash_node(LRUNode node, HashNode hash_node);
00083
00084 /**
00085  * @brief Obtiene el numero de bucket del hashnode asociado a `node`.
00086  *
00087  * @param node Puntero al nodo LRU.
00088  * @return El nmero de bucket del hash node asociado a `node`.
00089  */
00090
00091 */

```



```

00082 unsigned int lrunode_get_bucket_number(LRUNode node);
00083
00084 /**
00085  * @brief Establece el numero de bucket del nodo objetivo.
00086  * @param node Puntero al nodo LRU.
00087  * @param bucket_number Numero de bucket del hash node asociado a `node`.
00088  */
00089 void lrunode_set_bucket_number(LRUNode node, unsigned int bucket_number);
00090
00091 #endif // __LRU_NODE_H__

```

4.47 server/cache_server.c File Reference

```

#include "cache_server_utils.h"
#include <assert.h>
#include "../cache/cache.h"
#include "../helpers/quit.h"

```

Macros

- #define **GREEN** "\x1b[32m"
- #define **RED** "\x1b[31m"
- #define **ORANGE** "\x1b[38;5;214m"
- #define **BLUE** "\x1b[94m"
- #define **SOFT_RED** "\x1b[38;5;210m"
- #define **RESET** "\x1b[0m"

Functions

- void * **working_thread** (void *thread_args)
Es la funcion con la que se lanza a cada uno de los threads que responden pedidos por el servidor.
- void **start_server** (ServerArgs *server_args)
Recibe la informacion del servidor (file descriptor de la instancia epoll, file descriptor del socket del servidor, y cantidad de threads) y lanza los threads trabajadores correspondientes, comenzando asi el funcionamiento del servidor.
- int **main** (int argc, char **argv)
Pone en funcionamiento al servidor memcached con el socket, numero de threads, e instancia de la cache pasados por argumento. No se chequea la correctitud de los argumentos pues esta funcion solo sera invocada por server_↵ starter que no permite errores.

4.47.1 Macro Definition Documentation

4.47.1.1 BLUE

```
#define BLUE "\x1b[94m"
```

Definition at line 11 of file [cache_server.c](#).

4.47.1.2 GREEN

```
#define GREEN "\x1b[32m"
```

Definition at line 8 of file [cache_server.c](#).

4.47.1.3 ORANGE

```
#define ORANGE "\x1b[38;5;214m"
```

Definition at line 10 of file [cache_server.c](#).

4.47.1.4 RED

```
#define RED "\x1b[31m"
```

Definition at line 9 of file [cache_server.c](#).

4.47.1.5 RESET

```
#define RESET "\x1b[0m"
```

Definition at line 15 of file [cache_server.c](#).

4.47.1.6 SOFT_RED

```
#define SOFT_RED "\x1b[38;5;210m"
```

Definition at line 12 of file [cache_server.c](#).

4.47.2 Function Documentation

4.47.2.1 main()

```
int main (
    int argc,
    char ** argv )
```

Pone en funcionamiento al servidor memcached con el socket, numero de threads, e instancia de la cache pasados por argumento. No se chequea la correctitud de los argumentos pues esta funcion solo sera invocada por `server_starter` que no permite errores.

Definition at line 203 of file [cache_server.c](#).

```
00203         { // Sabemos que los argumentos son correctos.
00204
00205     assert(argc == 3);
00206     setbuf(stdout, NULL); // Opcional
00207
00208     ServerArgs server_args;
00209     server_args.server_socket = atoi(argv[1]);
00210     server_args.num_threads = atoi(argv[2]);
00211     Cache global_cache = cache_create((HashFunction) kr_hash, server_args.num_threads);
00212
00213     if (global_cache == NULL) {
00214         fprintf(stderr, "[Error] Not enough memory to create the cache\n");
00215         abort();
00216     }
00217
00218     server_args.cache = global_cache;
00219
00220     start_server(&server_args);
00221
00222     return 0;
00223 }
```

4.47.2.2 start_server()

```
void start_server (
    ServerArgs * server_args )
```

Recibe la informacion del servidor (file descriptor de la instancia epoll, file descriptor del socket del servidor, y cantidad de threads) y lanza los threads trabajadores correspondientes, comenzando asi el funcionamiento del servidor.

Parameters

in	<i>server_args</i>	Puntero a la estructura que contiene la informacion de esta instancia del servidor.
----	--------------------	---

Definition at line 155 of file [cache_server.c](#).

```
00155     {
00156
00157     ThreadArgs thread_args;
00158
00159     thread_args.server_epoll = epoll_create1(0);
00160     if (thread_args.server_epoll < 0) quit("Error: failed to create the epoll instance");
00161
00162     thread_args.server_socket = server_args->server_socket;
00163     thread_args.cache = server_args->cache;
00164
00165     // El primer evento que controlamos es el de EPOLLIN al socket del servidor, que representa un
    intento de conexion. Solo cargamos el campo de socket, pues la informacion de parseo no aplica a este
    evento.
00166     ClientData server_data;
00167     server_data.socket = server_args->server_socket;
00168 }
```

```

00169     struct epoll_event event;
00170     event.events = EPOLLIN | EPOLLONESHOT;
00171
00172     // Al levantarse un thread, podra distinguir si se trata de un intento de conexion al chequear el
    socket que cargamos en server_data.
00173     event.data.ptr = &server_data;
00174     if (epoll_ctl(thread_args.server_epoll, EPOLL_CTL_ADD, server_data.socket, &event) < 0)
00175         quit("Error: failed to add server socket to the epoll instance");
00176
00177     // Leemos la cantidad de threads a lanzar y los creamos
00178     int num_threads = server_args->num_threads;
00179
00180     // Creamos un ThreadArgs por cada thread, donde cada uno es una copia del thread_args que creamos en
    un principio.
00181     pthread_t threads[num_threads];
00182     ThreadArgs threads_args[num_threads];
00183
00184     for (int i = 0 ; i < num_threads ; i++) {
00185         threads_args[i] = thread_args;
00186         threads_args[i].thread_number = i;
00187
00188         if (pthread_create(&threads[i], NULL, working_thread, &threads_args[i]) != 0)
00189             quit("Error: failed to create a thread of the server");
00190
00191     }
00192
00193     for (int i = 0 ; i < num_threads ; i++) {
00194         if (pthread_join(threads[i], NULL) != 0)
00195             quit("Error: failed to join a thread of the server");
00196     }
00197 }

```

4.47.2.3 working_thread()

```

void * working_thread (
    void * thread_args )

```

Es la funcion con la que se lanza a cada uno de los threads que responden pedidos por el servidor.

Cada thread entra en un ciclo infinito en el que

- Espera a ser levantado por la instancia de epoll asociada al servidor.
- Al levantarse, recupera la informacion del cliente y evento que lo provoco. Aqui se obtiene `event_data`, la estructura que contiene el socket asociado al cliente que intenta realizar una operacion de IO y su buffer mas la informacion de parseo.
- Si el socket del `event_data` es el socket del servidor, entonces se trata de un intento de conexion. El thread acepta la conexion, crea la estructura de datos para el nuevo cliente, reconstruye la instancia de epoll y vuelve a iniciar el ciclo.
- Si el socket del `event_data` es distinto al del servidor, se trata de un cliente ya conectado intentando escribir. Se parsea el stream recibido y actualiza la informacion del cliente. Si el parseo llego a su estado final, se ejecuta el pedido. En todos los casos, se reconstruye la instancia de epoll y se vuelve a iniciar el ciclo.

Definition at line 62 of file `cache_server.c`.

```

00062     {
00063
00064         ThreadArgs* thread_args_casted = (ThreadArgs*) thread_args;
00065
00066         int server_epoll = thread_args_casted->server_epoll;
00067         int server_socket = thread_args_casted->server_socket;
00068         int thread_number = thread_args_casted->thread_number;
00069         Cache cache = thread_args_casted->cache;
00070
00071         struct epoll_event event;
00072
00073         while(1) {
00074
00075             print_waiting_msg(thread_number);
00076             if (epoll_wait(server_epoll, &event, 1, -1) < 0)
00077                 quit("[Error] EPOLL_WAIT");
00078
00079             ClientData* event_data = (ClientData*) event.data.ptr;
00080
00081             if (event.events & EPOLLERR) {
00082                 print_error_msg(thread_number);
00083                 drop_client(server_epoll, event_data);
00084             }
00085         }
00086     }

```

```

00085
00086     else if (event.events & (EPOLLHUP | EPOLLRDHUP)) {
00087         print_disconnect_msg(thread_number);
00088         drop_client(server_epoll, event_data);
00089     }
00090
00091     else if (event_data->socket == server_socket) { // Alguien se quiere conectar
00092
00093         int new_client_socket = accept(server_socket, NULL, NULL);
00094         print_accepted_msg(thread_number);
00095
00096         // Creamos el epoll_event del cliente y lo cargamos al epoll
00097         ClientData* new_client_data = create_new_client_data(new_client_socket, cache);
00098
00099         // Si fallo la creacion del nuevo cliente,
00100         if (new_client_data == NULL && new_client_socket >= 0) {
00101             // Le cerramos el socket
00102             close(new_client_socket);
00103         }
00104         else {
00105             // Si pude crear el cliente, lo cargo al epoll
00106             construct_new_client_epoll(server_epoll, new_client_data);
00107         }
00108
00109         // Reconstruimos el evento de epoll del servidor
00110         event.events = EPOLLIN | EPOLLONESHOT;
00111         epoll_ctl(server_epoll, EPOLL_CTL_MOD, server_socket, &event);
00112     }
00113 }
00114
00115 else { // Es un cliente escribiendo
00116
00117     // Parseamos su request
00118     print_parsing_msg(thread_number);
00119
00120     // Si fracaso el parseo, droppeamos al cliente
00121     if (parse_request(event_data, cache) < 0) {
00122         print_error_msg(thread_number);
00123         drop_client(server_epoll, event_data);
00124         continue;
00125     }
00126
00127     // Si el parseo termino, ejecutamos su pedido
00128     if (event_data->parsing_stage == PARSING_FINISHED) {
00129         print_handling_msg(thread_number);
00130
00131         // Si fracaso el envio de la respuesta al cliente, lo droppeamos
00132         if (handle_request(event_data, cache) < 0) {
00133             print_error_msg(thread_number);
00134             drop_client(server_epoll, event_data);
00135         }
00136
00137         // Si no, reiniciamos sus buffers y reconstruimos su evento en el epoll.
00138         reset_client_data(event_data);
00139     }
00140 }
00141
00142 reconstruct_client_epoll(server_epoll, &event, event_data);
00143
00144 // Caso contrario, continua la proxima iteracion.
00145 }
00146 }
00147 }
00148 }

```

4.48 cache_server.c

[Go to the documentation of this file.](#)

```

00001
00002 #include "cache_server_utils.h"
00003 #include <assert.h>
00004
00005 #include "../cache/cache.h"
00006 #include "../helpers/quit.h"
00007
00008 #define GREEN    "\x1b[32m"
00009 #define RED      "\x1b[31m"
00010 #define ORANGE   "\x1b[38;5;214m"
00011 #define BLUE     "\x1b[94m"
00012 #define SOFT_RED "\x1b[38;5;210m"
00013
00014
00015 #define RESET    "\x1b[0m"
00016

```

```

00017 static void print_error_msg(int thread_number) {
00018     printf(RED "[Thread %d] " RESET, thread_number);
00019     printf("Error en socket\n");
00020 }
00021
00022 static void print_disconnect_msg(int thread_number) {
00023     printf(SOFT_RED "[Thread %d] " RESET, thread_number);
00024     printf("Client disconnected.\n");
00025 }
00026
00027 static void print_waiting_msg(int thread_number) {
00028     printf(GREEN "[Thread %d] " RESET, thread_number);
00029     printf("Waiting for events.\n");
00030 }
00031
00032 static void print_accepted_msg(int thread_number) {
00033     printf(ORANGE "[Thread %d] " RESET, thread_number);
00034     printf("Accepting client.\n");
00035 }
00036
00037 static void print_parsing_msg(int thread_number) {
00038     printf(BLUE "[Thread %d] " RESET, thread_number);
00039     printf("Parsing request.\n");
00040 }
00041
00042 static void print_handling_msg(int thread_number) {
00043     printf(BLUE "[Thread %d] " RESET, thread_number);
00044     printf("Handling request.\n");
00045 }
00046
00047
00048
00049 /**
00050  * @brief Es la funcion con la que se lanza a cada uno de los threads que responden pedidos por el
00051  * servidor.
00052  * Cada thread entra en un ciclo infinito en el que
00053  *
00054  * - Espera a ser levantado por la instancia de epoll asociada al servidor.
00055  * - Al levantarse, recupera la informacion del cliente y evento que lo provoco. Aqui se obtiene
00056  * `event_data`, la estructura que contiene el socket asociado al cliente que intenta realizar una
00057  * operacion de IO y su buffer mas la informacion de parseo.
00058  *
00059  * - Si el socket del `event_data` es el socket del servidor, entonces se trata de un intento de
00060  * conexon. El thread acepta la conexon, crea la estructura de datos para el nuevo cliente, reconstruye
00061  * la instancia de epoll y vuelve a iniciar el ciclo.
00062  *
00063  * - Si el socket del `event_data` es distinto al del servidor, se trata de un cliente ya conectado
00064  * intentando escribir. Se parsea el stream recibido y actualiza la informacion del cliente. Si el parseo
00065  * llego a su estado final, se ejecuta el pedido. En todos los casos, se reconstruye la instancia de
00066  * epoll y se vuelve a iniciar el ciclo.
00067  */
00068 void* working_thread(void* thread_args) {
00069     ThreadArgs* thread_args_casted = (ThreadArgs*) thread_args;
00070
00071     int server_epoll = thread_args_casted->server_epoll;
00072     int server_socket = thread_args_casted->server_socket;
00073     int thread_number = thread_args_casted->thread_number;
00074     Cache cache = thread_args_casted->cache;
00075
00076     struct epoll_event event;
00077
00078     while(1) {
00079         print_waiting_msg(thread_number);
00080         if (epoll_wait(server_epoll, &event, 1, -1) < 0)
00081             quit("[Error] EPOLL_WAIT");
00082
00083         ClientData* event_data = (ClientData*) event.data.ptr;
00084
00085         if (event.events & EPOLLERR) {
00086             print_error_msg(thread_number);
00087             drop_client(server_epoll, event_data);
00088         }
00089
00090         else if (event.events & (EPOLLHUP | EPOLLRDHUP)) {
00091             print_disconnect_msg(thread_number);
00092             drop_client(server_epoll, event_data);
00093         }
00094
00095         else if (event_data->socket == server_socket) { // Alguien se quiere conectar
00096             int new_client_socket = accept(server_socket, NULL, NULL);
00097             print_accepted_msg(thread_number);
00098         }
00099     }
00100 }

```

```

00096         // Creamos el epoll_event del cliente y lo cargamos al epoll
00097         ClientData* new_client_data = create_new_client_data(new_client_socket, cache);
00098
00099         // Si fallo la creacion del nuevo cliente,
00100         if (new_client_data == NULL && new_client_socket >= 0) {
00101             // Le cerramos el socket
00102             close(new_client_socket);
00103         }
00104         else {
00105             // Si pude crear el cliente, lo cargo al epoll
00106             construct_new_client_epoll(server_epoll, new_client_data);
00107         }
00108
00109         // Reconstruimos el evento de epoll del servidor
00110         event.events = EPOLLIN | EPOLLONESHOT;
00111         epoll_ctl(server_epoll, EPOLL_CTL_MOD, server_socket, &event);
00112     }
00113 }
00114
00115 else { // Es un cliente escribiendo
00116
00117     // Parseamos su request
00118     print_parsing_msg(thread_number);
00119
00120     // Si fracaso el parseo, droppeamos al cliente
00121     if (parse_request(event_data, cache) < 0) {
00122         print_error_msg(thread_number);
00123         drop_client(server_epoll, event_data);
00124         continue;
00125     }
00126
00127     // Si el parseo termino, ejecutamos su pedido
00128     if (event_data->parsing_stage == PARSING_FINISHED) {
00129         print_handling_msg(thread_number);
00130
00131         // Si fracaso el envio de la respuesta al cliente, lo droppeamos
00132         if (handle_request(event_data, cache) < 0) {
00133             print_error_msg(thread_number);
00134             drop_client(server_epoll, event_data);
00135         }
00136
00137         // Si no, reiniciamos sus buffers y reconstruimos su evento en el epoll.
00138         reset_client_data(event_data);
00139     }
00140 }
00141
00142 reconstruct_client_epoll(server_epoll, &event, event_data);
00143
00144 // Caso contrario, continua la proxima iteracion.
00145 }
00146 }
00147 }
00148 }
00149
00150 /**
00151  * @brief Recibe la informacion del servidor (file descriptor de la instancia epoll, file
00152  * descriptor del socket del servidor, y cantidad de threads) y lanza los threads trabajadores
00153  * correspondientes, comenzando asi el funcionamiento del servidor.
00154  *
00155  * @param[in] server_args Puntero a la estructura que contiene la informacion de esta instancia del
00156  * servidor.
00157  */
00158 void start_server(ServerArgs* server_args) {
00159     ThreadArgs thread_args;
00160
00161     thread_args.server_epoll = epoll_create1(0);
00162     if (thread_args.server_epoll < 0) quit("Error: failed to create the epoll instance");
00163
00164     thread_args.server_socket = server_args->server_socket;
00165     thread_args.cache = server_args->cache;
00166
00167     // El primer evento que controlamos es el de EPOLLIN al socket del servidor, que representa un
00168     // intento de conexion. Solo cargamos el campo de socket, pues la informacion de parseo no aplica a este
00169     // evento.
00170     ClientData server_data;
00171     server_data.socket = server_args->server_socket;
00172
00173     struct epoll_event event;
00174     event.events = EPOLLIN | EPOLLONESHOT;
00175
00176     // Al levantarse un thread, podra distinguir si se trata de un intento de conexion al chequear el
00177     // socket que cargamos en server_data.
00178     event.data.ptr = &server_data;
00179     if (epoll_ctl(thread_args.server_epoll, EPOLL_CTL_ADD, server_data.socket, &event) < 0)
00180         quit("Error: failed to add server socket to the epoll instance");
00181 }

```

```

00177 // Leemos la cantidad de threads a lanzar y los creamos
00178 int num_threads = server_args->num_threads;
00179
00180 // Creamos un ThreadArgs por cada thread, donde cada uno es una copia del thread_args que creamos en
un principio.
00181 pthread_t threads[num_threads];
00182 ThreadArgs threads_args[num_threads];
00183
00184 for (int i = 0 ; i < num_threads ; i++) {
00185     threads_args[i] = thread_args;
00186     threads_args[i].thread_number = i;
00187
00188     if (pthread_create(&threads[i], NULL, working_thread, &threads_args[i]) != 0)
00189         quit("Error: failed to create a thread of the server");
00190
00191 }
00192
00193 for (int i = 0 ; i < num_threads ; i++) {
00194     if (pthread_join(threads[i], NULL) != 0)
00195         quit("Error: failed to join a thread of the server");
00196 }
00197 }
00198
00199
00200 /**
00201  * @brief Pone en funcionamiento al servidor memcached con el socket, numero de threads, e instancia
de la cache pasados por argumento. No se chequea la correctitud de los argumentos pues esta funcion
solo sera invocada por server_starter que no permite errores.
00202  */
00203 int main(int argc, char** argv) { // Sabemos que los argumentos son correctos.
00204
00205     assert(argc == 3);
00206     setbuf(stdout, NULL); // Opcional
00207
00208     ServerArgs server_args;
00209     server_args.server_socket = atoi(argv[1]);
00210     server_args.num_threads = atoi(argv[2]);
00211     Cache global_cache = cache_create((HashFunction) kr_hash, server_args.num_threads);
00212
00213     if (global_cache == NULL) {
00214         fprintf(stderr, "[Error] Not enough memory to create the cache\n");
00215         abort();
00216     }
00217
00218     server_args.cache = global_cache;
00219
00220     start_server(&server_args);
00221
00222     return 0;
00223 }
00224
00225
00226
00227

```

4.49 server/cache_server_models.h File Reference

Classes

- struct [ThreadArgs](#)
- struct [ServerArgs](#)
- struct [ClientData](#)

Macros

- #define [LENGTH_PREFIX_SIZE](#) 4

Enumerations

- enum [ParsingStage](#) {
[PARSING_COMMAND](#) , [PARSING_KEY_LEN](#) , [PARSING_KEY](#) , [PARSING_VALUE_LEN](#) ,
[PARSING_VALUE](#) , [PARSING_FINISHED](#) }
- enum [Command](#) { [PUT](#) = 11 , [DEL](#) = 12 , [GET](#) = 13 , [STATS](#) = 21 }
- enum [Response](#) {
[OKAY](#) = 101 , [EINVAL](#) = 111 , [ENOTFOUND](#) = 112 , [EBINARY](#) = 113 ,
[EBIG](#) = 114 , [EUNK](#) = 115 }

4.49.1 Macro Definition Documentation

4.49.1.1 LENGTH_PREFIX_SIZE

```
#define LENGTH_PREFIX_SIZE 4
```

Definition at line 4 of file [cache_server_models.h](#).

4.49.2 Enumeration Type Documentation

4.49.2.1 Command

```
enum Command
```

Enumerator

PUT	
DEL	
GET	
STATS	

Definition at line 17 of file [cache_server_models.h](#).

```
00017     {
00018
00019     PUT      = 11,
00020     DEL      = 12,
00021     GET      = 13,
00022     STATS    = 21
00023
00024 } Command;
```

4.49.2.2 ParsingStage

```
enum ParsingStage
```

Enumerator

PARSING_COMMAND	
PARSING_KEY_LEN	
PARSING_KEY	
PARSING_VALUE_LEN	
PARSING_VALUE	
PARSING_FINISHED	

Definition at line 6 of file [cache_server_models.h](#).

```
00006     {
00007
00008     PARSING_COMMAND,
00009     PARSING_KEY_LEN,
00010     PARSING_KEY,
00011     PARSING_VALUE_LEN,
00012     PARSING_VALUE,
00013     PARSING_FINISHED
00014
00015 } ParsingStage;
```

4.49.2.3 Response

```
enum Response
```

Enumerator

OKAY	
EINVALID	
ENOTFOUND	
EBINARY	

Enumerator

EBIG	
EUNK	

Definition at line 27 of file [cache_server_models.h](#).

```

00027     {
00028
00029         OKAY          = 101,
00030         EINVALID      = 111,
00031         ENOTFOUND     = 112,
00032         EBINARY       = 113,
00033         EBIG          = 114,
00034         EUNK          = 115
00035
00036     } Response;

```

4.50 cache_server_models.h

[Go to the documentation of this file.](#)

```

00001 #ifndef __CACHE_SERVER_MODELS_H__
00002 #define __CACHE_SERVER_MODELS_H__
00003
00004 #define LENGTH_PREFIX_SIZE 4
00005
00006 typedef enum {
00007
00008     PARSING_COMMAND,
00009     PARSING_KEY_LEN,
00010     PARSING_KEY,
00011     PARSING_VALUE_LEN,
00012     PARSING_VALUE,
00013     PARSING_FINISHED
00014 } ParsingStage;
00015
00016
00017 typedef enum {
00018
00019     PUT          = 11,
00020     DEL          = 12,
00021     GET          = 13,
00022     STATS        = 21
00023
00024 } Command;
00025
00026
00027 typedef enum {
00028
00029     OKAY          = 101,
00030     EINVALID      = 111,
00031     ENOTFOUND     = 112,
00032     EBINARY       = 113,
00033     EBIG          = 114,
00034     EUNK          = 115
00035
00036 } Response;
00037
00038 typedef struct {
00039
00040     int server_epoll;
00041     int server_socket;
00042     int thread_number;
00043     Cache cache;
00044
00045 } ThreadArgs;
00046
00047 typedef struct {
00048
00049     int server_epoll;
00050     int server_socket;
00051     int num_threads;
00052     Cache cache;
00053
00054 } ServerArgs;
00055
00056 typedef struct {
00057
00058     int socket;
00059
00060     char command;
00061

```

```

00062 char key_size_buffer[LENGTH_PREFIX_SIZE];
00063 int key_size;
00064 char* key;
00065
00066 char value_size_buffer[LENGTH_PREFIX_SIZE];
00067 int value_size;
00068 char* value;
00069
00070 int parsing_index;
00071 ParsingStage parsing_stage;
00072
00073 int cleaning;
00074
00075 } ClientData;
00076
00077
00078
00079 #endif // __CACHE_SERVER_MODELS_H__

```

4.51 server/cache_server_utils.c File Reference

```

#include <sys/epoll.h>
#include <stdio.h>
#include "cache_server_utils.h"
#include "cache_server_models.h"
#include "../cache/cache.h"

```

Macros

- #define `TRASH_BUFFER_SIZE` 100

Functions

- `ssize_t clean_socket (ClientData *client, int size)`
Consume hasta size bytes del client actualizando acordemente el indice de parseo del ClientData. No guarda la informacion.
- `ssize_t recv_client (ClientData *client, char *message_buffer, int size)`
Lee hasta size bytes del client en message_buffer actualizando acordemente el indice de parseo del ClientData.
- `ssize_t send_client (ClientData *client, char *message, int size)`
Escribe size bytes del mensaje message al client objetivo.
- `int parse_request (ClientData *cdata, Cache cache)`
Parsea el mensaje proveniente del cliente asociado al data de acuerdo a su parsing stage. Este parseo siempre es total; en caso de recibirse un mensaje incompleto, la informacion del cliente se actualiza y se vuelve a invocar la funcion recursivamente hasta que se complete el mensaje o se produzca un error.
- `int handle_request (ClientData *cdata, Cache cache)`
Ejecuta el comando cargado en la estructura con informacion del cliente apuntada por cdata en su campo command. Tambien libera la memoria asignada al buffer dinamico donde se almacena la key en el cdata, en caso de ser necesario.
- `void reset_client_data (ClientData *cdata)`
Restablece la estructura de informacion del cliente apuntada por data, inicializando a 0 los valores asociados al stage de parseo.
- `int reconstruct_client_epoll (int epoll_fd, struct epoll_event *ev, ClientData *cdata)`
Reconstruye el epoll_event asociado al cliente cuya informacion se almacena en la estructura apuntada por data, y carga el evento en la instancia de epoll asociada a epoll_fd para su control.
- `int construct_new_client_epoll (int epoll_fd, ClientData *cdata)`
Reconstruye el epoll_event asociado al cliente cuya informacion se almacena en la estructura apuntada por data, y carga el evento en la instancia de epoll asociada a epoll_fd para su control.
- `ClientData * create_new_client_data (int client_socket, Cache cache)`
Crea e inicializa una nueva estructura de informacion del cliente con el socket de cliente establecido a client_↔ socket.

- void `delete_client_data` (`ClientData` *cdata)
Destruye la estructura de informacion del cliente objetivo.
- void `drop_client` (int `epoll_fd`, `ClientData` *cdata)
Desconecta a un cliente de la instancia `epoll` y destruye su estructura de cliente asociada.

4.51.1 Macro Definition Documentation

4.51.1.1 TRASH_BUFFER_SIZE

#define TRASH_BUFFER_SIZE 100

Definition at line 7 of file `cache_server_utils.c`.

4.51.2 Function Documentation

4.51.2.1 clean_socket()

```
ssize_t clean_socket (
    ClientData * client,
    int size )
```

Consume hasta `size` bytes del `client` actualizando acordemente el indice de parseo del `ClientData`. No guarda la informacion.

Parameters

out	<code>client</code>	Estructura de datos del cliente.
	<code>size</code>	Cantidad maxima de bytes a consumir.

Returns

La cantidad de bytes efectivamente consumidos, o -1 si se produjo un error.

Definition at line 9 of file `cache_server_utils.c`.

```
00009                                     {
00010
00011     int socket = client->socket;
00012
00013     ssize_t total_bytes_received = 0;
00014     ssize_t bytes_received;
00015
00016     char buffer[TRASH_BUFFER_SIZE];
00017
00018     do {
00019
00020         bytes_received = recv(socket, buffer, TRASH_BUFFER_SIZE, 0);
00021
00022         total_bytes_received += bytes_received;
00023
00024     } while (bytes_received > 0 && total_bytes_received < size);
00025
00026     if (bytes_received < 0) {
00027         perror("Error: failed to call recv bytes.");
00028         return -1;
00029     }
00030
00031     client->parsing_index += total_bytes_received;
00032
00033     return total_bytes_received;
00034 }
```

4.51.2.2 construct_new_client_epoll()

```
int construct_new_client_epoll (
    int epoll_fd,
    ClientData * cdata )
```

Reconstruye el `epoll_event` asociado al cliente cuya informacion se almacena en la estructura apuntada por `data`, y carga el evento en la instancia de `epoll` asociada a `epoll_fd` para su control.

Parameters

<i>epoll↵ _fd</i>	File descriptor de la instancia de epoll objetivo.
<i>data</i>	Puntero a la estructura de informacion del cliente a controlar.

Returns

0 si la manipulacion de la instancia de epoll es exitosa, -1 si no.

Definition at line 356 of file [cache_server_utils.c](#).

```
00356                                     {
00357
00358     struct epoll_event ev;
00359
00360     ev.events    = EPOLLIN | EPOLLRDHUP | EPOLLONESHOT;
00361     ev.data.ptr = cdata;
00362
00363     return epoll_ctl(epoll_fd, EPOLL_CTL_ADD, cdata->socket, &ev);
00364
00365 }
```

4.51.2.3 create_new_client_data()

```
ClientData * create_new_client_data (
    int client_socket,
    Cache cache )
```

Crea e inicializa una nueva estructura de informacion del cliente con el socket de cliente establecido a *client↵
_socket*.

Parameters

<i>client_socket</i>	File descriptor del socket del cliente.
<i>cache</i>	Puntero a la cache con la que se comunicara el nuevo cliente.

Returns

Un puntero a la estructura creada e inicializada.

Definition at line 368 of file [cache_server_utils.c](#).

```
00368                                     {
00369
00370     ClientData* new_cdata = dynalloc(sizeof(ClientData), cache);
00371     if (new_cdata == NULL)
00372         return NULL;
00373
00374     memset(new_cdata->key_size_buffer, 0, LENGTH_PREFIX_SIZE);
00375     memset(new_cdata->value_size_buffer, 0, LENGTH_PREFIX_SIZE);
00376
00377     new_cdata->parsing_index = 0;
00378     new_cdata->parsing_stage = PARSING_COMMAND;
00379
00380     new_cdata->socket = client_socket;
00381
00382     new_cdata->cleaning = 0;
00383
00384     new_cdata->key = NULL;
00385     new_cdata->value = NULL;
00386
00387     return new_cdata;
00388 }
```

4.51.2.4 delete_client_data()

```
void delete_client_data (
    ClientData * cdata )
```

Destruye la estructura de informacion del cliente objetivo.

Parameters

<i>cdata</i>	El puntero a la estructura a destruir.
--------------	--

Definition at line 391 of file `cache_server_utils.c`.

```
00391
00392
00393     if (cdata == NULL)
00394         return;
00395
00396     if (cdata->socket > 0)
00397         close(cdata->socket);
00398
00399     free(cdata);
00400
00401 }
```

4.51.2.5 drop_client()

```
void drop_client (
    int epoll_fd,
    ClientData * cdata )
```

Desconecta a un cliente de la instancia epoll y destruye su estructura de cliente asociada.

Parameters

	<i>epoll_fd</i>	File descriptor de la instancia de epoll donde se monitoreaba al cliente.
out	<i>cdata</i>	Puntero a la estructura del cliente a eliminar.

Definition at line 404 of file `cache_server_utils.c`.

```
00404
00405     epoll_ctl(epoll_fd, EPOLL_CTL_DEL, cdata->socket, NULL);
00406     delete_client_data(cdata);
00407 }
```

4.51.2.6 handle_request()

```
int handle_request (
    ClientData * cdata,
    Cache cache )
```

Ejecuta el comando cargado en la estructura con informacion del cliente apuntada por *cdata* en su campo *command*. Tambien libera la memoria asignada al buffer dinamico donde se almacena la *key* en el *cdata*, en caso de ser necesario.

Parameters

	<i>cdata</i>	Puntero a la estructura con informacion del cliente.
in	<i>cache</i>	El puntero a la cache asociada al pedido a parsear.

Returns

0 si la ejecucion del pedido es exitosa, -1 si se produjo un error al intentar enviar la respuesta.

Definition at line 223 of file `cache_server_utils.c`.

```
00223
00224
00225     // En todo momento, si hay un problema con el socket devolvemos -1. Del lado del server, esto hace
    que se droppee al cliente.
00226
00227     char command;
00228
00229     switch (cdata->command) {
00230
00231     case PUT:
00232
00233         int put_status = cache_put(cdata->key, cdata->key_size,
```

```

00234             cdata->value, cdata->value_size, cache);
00235
00236         // Si el put fue exitoso, respondemos con OK, si no, con EUNK
00237         command = put_status < 0 ? ENOTFOUND : OKAY;
00238
00239         // Si lo que se hizo fue pisar el valor, hay que liberar la memoria de la key, pues queda la
anterior
00240         if (put_status == 1)
00241             free(cdata->key);
00242
00243         // Si se produjo un error, hay que liberar tanto la clave como el valor, pues ninguna de las dos
fue insertada a la cache.
00244         else if (put_status < 0) {
00245             free(cdata->key);
00246             free(cdata->value);
00247         }
00248
00249         if (send_client(cdata, &command, 1) < 0) return -1;
00250
00251         break;
00252
00253     case DEL:
00254
00255         int del_status = cache_delete(cdata->key, cdata->key_size, cache);
00256
00257         command = del_status == 0 ? OKAY :
00258             (del_status == 1 ? ENOTFOUND : ENOTFOUND);
00259
00260         // En cualquier caso, quiero liberar el buffer donde guarde la key una vez terminada la
operacion.
00261         if (cdata->key)
00262             free(cdata->key);
00263
00264         if (send_client(cdata, &command, 1) < 0) return -1;
00265
00266         break;
00267
00268     case GET:
00269
00270         LookupResult lr = cache_get(cdata->key, cdata->key_size, cache);
00271
00272         // En cualquier caso, quiero liberar el buffer donde guarde la key una vez terminada la
operacion.
00273         if (cdata->key)
00274             free(cdata->key);
00275
00276         if (lookup_result_is_ok(lr)) {
00277
00278             char command = OKAY;
00279             char length_prefix_buffer[LENGTH_PREFIX_SIZE];
00280             size_t size = ntohl(lookup_result_get_size(lr));
00281             memcpy(length_prefix_buffer, &size, LENGTH_PREFIX_SIZE);
00282
00283             if (send_client(cdata, &command, 1) < 0) return -1;
00284             if (send_client(cdata, length_prefix_buffer, LENGTH_PREFIX_SIZE) < 0) return -1;
00285             if (send_client(cdata,
00286                 lookup_result_get_value(lr),
00287                 lookup_result_get_size(lr)) < 0) return -1;
00288         }
00289
00290         else {
00291             command = ENOTFOUND;
00292             if (send_client(cdata, &command, 1) < 0) return -1;
00293         }
00294         break;
00295
00296     case STATS: {
00297
00298         // Obtengo el reporte de estadísticas.
00299         StatsReport report = cache_report(cache);
00300
00301         // Creo un buffer donde cargar el mensaje y lo llenamos
00302         size_t buffer_size = sizeof(char) * STATS_MESSAGE_LENGTH;
00303         char report_buffer[buffer_size];
00304         char command = OKAY;
00305
00306         int report_len = stats_report_stringify(report, report_buffer);
00307
00308         char length_prefix_buffer[LENGTH_PREFIX_SIZE];
00309         size_t size = ntohl(report_len);
00310         memcpy(length_prefix_buffer, &size, LENGTH_PREFIX_SIZE);
00311
00312         // Enviamos el comando el mensaje
00313         send_client(cdata, &command, 1); // Mando STATS
00314         send_client(cdata, length_prefix_buffer, LENGTH_PREFIX_SIZE); // Prefijo longitud
00315         send_client(cdata, report_buffer, report_len); // String del report
00316

```

```

00317         break;
00318     }
00319 }
00320
00321 case EBIG:
00322
00323     command = EBIG;
00324     if (send_client(cdata, &command, 1) < 0) return -1;
00325
00326     break;
00327 }
00328 }
00329
00330 return 0;
00331
00332 }
```

4.51.2.7 parse_request()

```

int parse_request (
    ClientData * cdata,
    Cache cache )
```

Parsea el mensaje proveniente del cliente asociado al `data` de acuerdo a su parsing stage. Este parseo siempre es total; en caso de recibirse un mensaje incompleto, la informacion del cliente se actualiza y se vuelve a invocar la funcion recursivamente hasta que se complete el mensaje o se produzca un error.

Parameters

out	<i>cdata</i>	Puntero a la estructura con la informacion del cliente que es actualizada de acuerdo a la informacion parseada.
in	<i>cache</i>	El puntero a la cache asociada al pedido a parsear.

Returns

0 si parseo correctamente, -1 si fracaso al recibir bytes del socket.

Definition at line 95 of file `cache_server_utils.c`.

```

00095     {
00096
00097     switch (cdata->parsing_stage) {
00098
00099     case PARSING_COMMAND:
00100
00101         if (recv_client(cdata, &cdata->command, 1) < 0) return -1;
00102
00103         // No se termino de parsear el comando
00104         if (cdata->parsing_index < 1) return 0;
00105
00106         // Si el comando no es valido, devolvemos -1 y se le cierra la conexion
00107         if (!command_is_valid(cdata->command)) return -1;
00108
00109         // Si no, determinamos el estado de parseo dependiendo de si es STATS o no
00110         cdata->parsing_stage = cdata->command == STATS ?
00111             PARSING_FINISHED : PARSING_KEY_LEN;
00112         cdata->parsing_index = 0;
00113
00114         break;
00115
00116     case PARSING_KEY_LEN:
00117
00118         if (recv_client(cdata,
00119             cdata->key_size_buffer + cdata->parsing_index,
00120             LENGTH_PREFIX_SIZE - cdata->parsing_index) < 0) return -1;
00121
00122         if (cdata->parsing_index < LENGTH_PREFIX_SIZE) return 0;
00123
00124         cdata->key_size = htonl(*(int*)(cdata->key_size_buffer));
00125
00126         cdata->key = dynalloc(cdata->key_size, cache);
00127         if (cdata->key == NULL) {
00128             // Si falla la asignacion de memoria, marcamos el comando como EBIG.
00129             // Pasamos al cliente a estado cleaning, vaciando todo lo que queda en el buffer para poder
00130             volver a sincronizarnos con el final del mensaje.
00131             cdata->command = EBIG;
00132             cdata->cleaning = 1;
00133         }
```

```

00134     cdata->parsing_stage = PARSING_KEY;
00135     cdata->parsing_index = 0;
00136
00137     break;
00138
00139     case PARSING_KEY:
00140
00141         if (cdata->cleaning) {
00142             if (clean_socket(cdata, cdata->key_size - cdata->parsing_index) < 0) {
00143                 return -1;
00144             }
00145         }
00146
00147         else if (recv_client(cdata, cdata->key + cdata->parsing_index,
00148                             cdata->key_size - cdata->parsing_index) < 0) {
00149             // Liberamos la memoria asignada hasta ahora y propagamos el error
00150             free(cdata->key);
00151             return -1;
00152         }
00153
00154         if (cdata->parsing_index < cdata->key_size) return 0;
00155
00156         cdata->parsing_stage = cdata->command == PUT ?
00157                               PARSING_VALUE_LEN : PARSING_FINISHED;
00158         cdata->parsing_index = 0;
00159
00160         break;
00161
00162     case PARSING_VALUE_LEN:
00163
00164         if (recv_client(cdata,
00165                         cdata->value_size_buffer + cdata->parsing_index,
00166                         LENGTH_PREFIX_SIZE - cdata->parsing_index) < 0) {
00167             free(cdata->key);
00168             return -1;
00169         }
00170
00171         // El recv ya no puede leer mas bytes, pero no terminamos de leer el prefijo de longitud.
Retornamos 0, el cliente volvera a ser controlado por el epoll, y esperamos a que vuelvan a llegar
bytes para leerlos.
00172         if (cdata->parsing_index < LENGTH_PREFIX_SIZE) return 0;
00173
00174         cdata->value_size = htonl(*(int*)(cdata->value_size_buffer));
00175
00176         // Si no estaba limpiando, pido la memoria para value.
00177         if (!cdata->cleaning) {
00178
00179             cdata->value = dynalloc(cdata->value_size, cache);
00180
00181             if (cdata->value == NULL) {
00182                 // Analogo al caso de fallar en key, pero libero la memoria de key
00183                 free(cdata->key);
00184                 cdata->command = EBIG;
00185                 cdata->cleaning = 1;
00186             }
00187         }
00188
00189     }
00190
00191     cdata->parsing_stage = PARSING_VALUE;
00192     cdata->parsing_index = 0;
00193
00194     break;
00195
00196     case PARSING_VALUE:
00197
00198         if (cdata->cleaning) {
00199             if (clean_socket(cdata, cdata->key_size - cdata->parsing_index) < 0) {
00200                 return -1; // No hace falta liberar cdata->key y cdata->value nunca, porque si llegue en
cleaning a este punto ambos son NULL ya.
00201             }
00202         }
00203
00204         else if (recv_client(cdata, cdata->value + cdata->parsing_index,
00205                             cdata->value_size - cdata->parsing_index) < 0) return -1;
00206
00207         if (cdata->parsing_index < cdata->value_size) return 0;
00208
00209         cdata->parsing_stage = PARSING_FINISHED;
00210
00211         break;
00212
00213     case PARSING_FINISHED: // Imposible que llegue hasta aca, lanzamos un error
00214         quit("Error: switch case PARSING_FINISHED reached");
00215         break;
00216 }
00217 if (cdata->parsing_stage != PARSING_FINISHED)

```



```

00218     return parse_request(cdata, cache);
00219
00220     return 0;
00221 }

```

4.51.2.8 reconstruct_client_epoll()

```

int reconstruct_client_epoll (
    int epoll_fd,
    struct epoll_event * ev,
    ClientData * data )

```

Reconstruye el `epoll_event` asociado al cliente cuya informacion se almacena en la estructura apuntada por `data`, y carga el evento en la instancia de `epoll` asociada a `epoll_fd` para su control.

Parameters

	<i>epoll_fd</i>	File descriptor de la instancia de <code>epoll</code> objetivo.
out	<i>epoll_event</i>	Puntero a la estructura del <code>epoll_event</code> a reconstruir.
	<i>data</i>	Puntero a la estructura de informacion del cliente a controlar.

Returns

0 si la manipulacion de la instancia de `epoll` es exitosa, -1 si no.

Definition at line 346 of file `cache_server_utils.c`.

```

00346
00347
00348     ev->events = EPOLLIN | EPOLLRDHUP | EPOLLONESHOT;
00349     ev->data.ptr = cdata;
00350
00351     return epoll_ctl(epoll_fd, EPOLL_CTL_MOD, cdata->socket, ev);
00352
00353 }

```

4.51.2.9 recv_client()

```

ssize_t recv_client (
    ClientData * client,
    char * message_buffer,
    int size )

```

Lee hasta `size` bytes del `client` en `message_buffer` actualizando acordemente el indice de parseo del `ClientData`.

Parameters

out	<i>client</i>	Estructura de datos del cliente.
	<i>message_buffer</i>	Buffer de lectura.
	<i>size</i>	Cantidad maxima de bytes a leer.

Returns

La cantidad de bytes efectivamente leidos, o -1 si se produjo un error.

Definition at line 37 of file `cache_server_utils.c`.

```

00037
00038
00039     int socket = client->socket;
00040
00041     ssize_t total_bytes_received = 0;
00042     ssize_t bytes_received;
00043
00044     do {
00045
00046         bytes_received = recv(socket, message_buffer + total_bytes_received,
00047                             size - total_bytes_received, 0);
00048

```

```

00049     total_bytes_received += bytes_received;
00050
00051 } while (bytes_received > 0 && total_bytes_received < size);
00052
00053 if (bytes_received < 0) {
00054     perror("Error: failed to call recv bytes.");
00055     return -1;
00056 }
00057
00058 client->parsing_index += total_bytes_received;
00059
00060 return total_bytes_received;
00061 }

```

4.51.2.10 reset_client_data()

```

void reset_client_data (
    ClientData * data )

```

Restablece la estructura de informacion del cliente apuntada por *data*, inicializando a 0 los valores asociados al stage de parseo.

Parameters

out	<i>data</i>	Puntero a la estructura de informacion del cliente a restablecer.
-----	-------------	---

Definition at line 335 of file [cache_server_utils.c](#).

```

00335                                     {
00336
00337     cdata->parsing_index = 0;
00338     cdata->parsing_stage = PARSING_COMMAND;
00339     cdata->cleaning      = 0;
00340     cdata->key           = NULL;
00341     cdata->value         = NULL;
00342
00343 }

```

4.51.2.11 send_client()

```

ssize_t send_client (
    ClientData * client,
    char * message,
    int size )

```

Escribe *size* bytes del mensaje *message* al *client* objetivo.

Parameters

<i>client</i>	Estructura del cliente a escribir.
<i>message</i>	Buffer con el contenido del mensaje.
<i>size</i>	Cantidad de bytes a escribir.

Returns

La cantidad de bytes enviados, que es -1 si se produjo un error.

Definition at line 64 of file [cache_server_utils.c](#).

```

00064                                     {
00065
00066     int socket = client->socket;
00067
00068     ssize_t total_bytes_sent = 0;
00069     ssize_t bytes_sent;
00070
00071     do {
00072
00073         bytes_sent = send(socket, message + total_bytes_sent,
00074                         size - total_bytes_sent, 0);
00075
00076         total_bytes_sent += bytes_sent;
00077
00078

```

```

00079     } while (bytes_sent > 0 && total_bytes_sent < size);
00080
00081     if (bytes_sent < 0) {
00082         perror("Error: failed to call send bytes.");
00083         return -1;
00084     }
00085
00086     return total_bytes_sent;
00087
00088 }

```

4.52 cache_server_utils.c

[Go to the documentation of this file.](#)

```

00001 #include <sys/epoll.h>
00002 #include <stdio.h>
00003 #include "cache_server_utils.h"
00004 #include "cache_server_models.h"
00005 #include "../cache/cache.h"
00006
00007 #define TRASH_BUFFER_SIZE 100
00008
00009 ssize_t clean_socket(ClientData* client, int size) {
00010
00011     int socket = client->socket;
00012
00013     ssize_t total_bytes_received = 0;
00014     ssize_t bytes_received;
00015
00016     char buffer[TRASH_BUFFER_SIZE];
00017
00018     do {
00019
00020         bytes_received = recv(socket, buffer, TRASH_BUFFER_SIZE, 0);
00021
00022         total_bytes_received += bytes_received;
00023
00024     } while (bytes_received > 0 && total_bytes_received < size);
00025
00026     if (bytes_received < 0) {
00027         perror("Error: failed to call recv bytes.");
00028         return -1;
00029     }
00030
00031     client->parsing_index += total_bytes_received;
00032
00033     return total_bytes_received;
00034 }
00035
00036
00037 ssize_t recv_client(ClientData* client, char* message_buffer, int size) {
00038
00039     int socket = client->socket;
00040
00041     ssize_t total_bytes_received = 0;
00042     ssize_t bytes_received;
00043
00044     do {
00045
00046         bytes_received = recv(socket, message_buffer + total_bytes_received,
00047                               size - total_bytes_received, 0);
00048
00049         total_bytes_received += bytes_received;
00050
00051     } while (bytes_received > 0 && total_bytes_received < size);
00052
00053     if (bytes_received < 0) {
00054         perror("Error: failed to call recv bytes.");
00055         return -1;
00056     }
00057
00058     client->parsing_index += total_bytes_received;
00059
00060     return total_bytes_received;
00061 }
00062
00063
00064 ssize_t send_client(ClientData* client, char* message, int size) {
00065
00066     int socket = client->socket;
00067
00068     ssize_t total_bytes_sent = 0;
00069     ssize_t bytes_sent;
00070

```

```

00071
00072     do {
00073
00074         bytes_sent = send(socket, message + total_bytes_sent,
00075                           size - total_bytes_sent, 0);
00076
00077         total_bytes_sent += bytes_sent;
00078
00079     } while (bytes_sent > 0 && total_bytes_sent < size);
00080
00081     if (bytes_sent < 0) {
00082         perror("Error: failed to call send bytes.");
00083         return -1;
00084     }
00085
00086     return total_bytes_sent;
00087 }
00088
00089
00090
00091 static int command_is_valid(char cmd) {
00092     return cmd == PUT || cmd == GET || cmd == DEL || cmd == STATS;
00093 }
00094
00095 int parse_request(ClientData* cdata, Cache cache) {
00096     switch (cdata->parsing_stage) {
00097
00098     case PARSING_COMMAND:
00099
00100         if (recv_client(cdata, &cdata->command, 1) < 0) return -1;
00101
00102         // No se termino de parsear el comando
00103         if (cdata->parsing_index < 1) return 0;
00104
00105         // Si el comando no es valido, devolvemos -1 y se le cierra la conexion
00106         if (!command_is_valid(cdata->command)) return -1;
00107
00108         // Si no, determinamos el estado de parseo dependiendo de si es STATS o no
00109         cdata->parsing_stage = cdata->command == STATS ?
00110                               PARSING_FINISHED : PARSING_KEY_LEN;
00111         cdata->parsing_index = 0;
00112
00113         break;
00114
00115     case PARSING_KEY_LEN:
00116
00117         if (recv_client(cdata,
00118                        cdata->key_size_buffer + cdata->parsing_index,
00119                        LENGTH_PREFIX_SIZE - cdata->parsing_index) < 0) return -1;
00120
00121         if (cdata->parsing_index < LENGTH_PREFIX_SIZE) return 0;
00122
00123         cdata->key_size = htonl(*(int*)(cdata->key_size_buffer));
00124
00125         cdata->key = dynalloc(cdata->key_size, cache);
00126         if (cdata->key == NULL) {
00127             // Si falla la asignacion de memoria, marcamos el comando como EBIG.
00128             // Pasamos al cliente a estado cleaning, vaciando todo lo que queda en el buffer para poder
00129             volver a sincronizarnos con el final del mensaje.
00130             cdata->command = EBIG;
00131             cdata->cleaning = 1;
00132         }
00133
00134         cdata->parsing_stage = PARSING_KEY;
00135         cdata->parsing_index = 0;
00136
00137         break;
00138
00139     case PARSING_KEY:
00140
00141         if (cdata->cleaning) {
00142             if (clean_socket(cdata, cdata->key_size - cdata->parsing_index) < 0) {
00143                 return -1;
00144             }
00145         }
00146
00147         else if (recv_client(cdata, cdata->key + cdata->parsing_index,
00148                             cdata->key_size - cdata->parsing_index) < 0) {
00149             // Liberamos la memoria asignada hasta ahora y propagamos el error
00150             free(cdata->key);
00151             return -1;
00152         }
00153
00154         if (cdata->parsing_index < cdata->key_size) return 0;
00155
00156         cdata->parsing_stage = cdata->command == PUT ?

```

```

00157             PARSING_VALUE_LEN : PARSING_FINISHED;
00158     cdata->parsing_index = 0;
00159
00160     break;
00161
00162     case PARSING_VALUE_LEN:
00163
00164         if (recv_client(cdata,
00165             cdata->value_size_buffer + cdata->parsing_index,
00166             LENGTH_PREFIX_SIZE - cdata->parsing_index) < 0) {
00167             free(cdata->key);
00168             return -1;
00169         }
00170
00171         // El recv ya no puede leer mas bytes, pero no terminamos de leer el prefijo de longitud.
00172         Retornamos 0, el cliente volvera a ser controlado por el epoll, y esperamos a que vuelvan a llegar
00173         bytes para leerlos.
00174         if (cdata->parsing_index < LENGTH_PREFIX_SIZE) return 0;
00175
00176         cdata->value_size = htonl(*(int*)(cdata->value_size_buffer));
00177
00178         // Si no estaba limpiando, pido la memoria para value.
00179         if (!cdata->cleaning) {
00180             cdata->value = dynalloc(cdata->value_size, cache);
00181
00182             if (cdata->value == NULL) {
00183                 // Analogo al caso de fallar en key, pero libero la memoria de key
00184                 free(cdata->key);
00185                 cdata->command = EBIG;
00186                 cdata->cleaning = 1;
00187             }
00188         }
00189
00190         cdata->parsing_stage = PARSING_VALUE;
00191         cdata->parsing_index = 0;
00192
00193         break;
00194
00195     case PARSING_VALUE:
00196
00197         if (cdata->cleaning) {
00198             if (clean_socket(cdata, cdata->key_size - cdata->parsing_index) < 0) {
00199                 return -1; // No hace falta liberar cdata->key y cdata->value nunca, porque si llegue en
00200                 cleaning a este punto ambos son NULL ya.
00201             }
00202         }
00203         else if (recv_client(cdata, cdata->value + cdata->parsing_index,
00204             cdata->value_size - cdata->parsing_index) < 0) return -1;
00205
00206         if (cdata->parsing_index < cdata->value_size) return 0;
00207
00208         cdata->parsing_stage = PARSING_FINISHED;
00209
00210         break;
00211
00212     case PARSING_FINISHED: // Imposible que llegue hasta aca, lanzamos un error
00213         quit("Error: switch case PARSING_FINISHED reached");
00214         break;
00215     }
00216
00217     if (cdata->parsing_stage != PARSING_FINISHED)
00218         return parse_request(cdata, cache);
00219
00220     return 0;
00221 }
00222
00223 int handle_request(ClientData* cdata, Cache cache) {
00224
00225     // En todo momento, si hay un problema con el socket devolvemos -1. Del lado del server, esto hace
00226     // que se droppee al cliente.
00227
00228     char command;
00229
00230     switch (cdata->command) {
00231     case PUT:
00232
00233         int put_status = cache_put(cdata->key, cdata->key_size,
00234             cdata->value, cdata->value_size, cache);
00235
00236         // Si el put fue exitoso, respondemos con OK, si no, con EUNK
00237         command = put_status < 0 ? ENOTFOUND : OKAY;
00238
00239         // Si lo que se hizo fue pisar el valor, hay que liberar la memoria de la key, pues queda la

```

```

    anterior
00240     if (put_status == 1)
00241         free(cdata->key);
00242
00243     // Si se produjo un error, hay que liberar tanto la clave como el valor, pues ninguna de las dos
    fue insertada a la cache.
00244     else if (put_status < 0) {
00245         free(cdata->key);
00246         free(cdata->value);
00247     }
00248
00249     if (send_client(cdata, &command, 1) < 0) return -1;
00250
00251     break;
00252
00253     case DEL:
00254
00255         int del_status = cache_delete(cdata->key, cdata->key_size, cache);
00256
00257         command = del_status == 0 ? OKAY :
00258                 (del_status == 1 ? ENOTFOUND : ENOTFOUND);
00259
00260         // En cualquier caso, quiero liberar el buffer donde guarde la key una vez terminada la
    operacion.
00261         if (cdata->key)
00262             free(cdata->key);
00263
00264         if (send_client(cdata, &command, 1) < 0) return -1;
00265
00266         break;
00267
00268     case GET:
00269
00270         LookupResult lr = cache_get(cdata->key, cdata->key_size, cache);
00271
00272         // En cualquier caso, quiero liberar el buffer donde guarde la key una vez terminada la
    operacion.
00273         if (cdata->key)
00274             free(cdata->key);
00275
00276         if (lookup_result_is_ok(lr)) {
00277
00278             char command = OKAY;
00279             char length_prefix_buffer[LENGTH_PREFIX_SIZE];
00280             size_t size = ntohl(lookup_result_get_size(lr));
00281             memcpy(length_prefix_buffer, &size, LENGTH_PREFIX_SIZE);
00282
00283             if (send_client(cdata, &command, 1) < 0) return -1;
00284             if (send_client(cdata, length_prefix_buffer, LENGTH_PREFIX_SIZE) < 0) return -1;
00285             if (send_client(cdata,
00286                             lookup_result_get_value(lr),
00287                             lookup_result_get_size(lr)) < 0) return -1;
00288         }
00289
00290         else {
00291             command = ENOTFOUND;
00292             if (send_client(cdata, &command, 1) < 0) return -1;
00293         }
00294         break;
00295
00296     case STATS: {
00297
00298         // Obtengo el reporte de estadísticas.
00299         StatsReport report = cache_report(cache);
00300
00301         // Creo un buffer donde cargar el mensaje y lo llenamos
00302         size_t buffer_size = sizeof(char) * STATS_MESSAGE_LENGTH;
00303         char report_buffer[buffer_size];
00304         char command = OKAY;
00305
00306         int report_len = stats_report_stringify(report, report_buffer);
00307
00308         char length_prefix_buffer[LENGTH_PREFIX_SIZE];
00309         size_t size = ntohl(report_len);
00310         memcpy(length_prefix_buffer, &size, LENGTH_PREFIX_SIZE);
00311
00312         // Enviamos el comando el mensaje
00313         send_client(cdata, &command, 1); // Mando STATS
00314         send_client(cdata, length_prefix_buffer, LENGTH_PREFIX_SIZE); // Prefijo longitud
00315         send_client(cdata, report_buffer, report_len); // String del report
00316
00317         break;
00318
00319     }
00320
00321     case EBIG:
00322

```

```
00323     command = EBIG;
00324     if (send_client(cdata, &command, 1) < 0) return -1;
00325
00326     break;
00327 }
00328 }
00329
00330 return 0;
00331
00332 }
00333
00334
00335 void reset_client_data(ClientData* cdata) {
00336
00337     cdata->parsing_index = 0;
00338     cdata->parsing_stage = PARSING_COMMAND;
00339     cdata->cleaning = 0;
00340     cdata->key = NULL;
00341     cdata->value = NULL;
00342 }
00343 }
00344
00345
00346 int reconstruct_client_epoll(int epoll_fd, struct epoll_event* ev, ClientData* cdata) {
00347
00348     ev->events = EPOLLIN | EPOLLRDHUP | EPOLLONESHOT;
00349     ev->data.ptr = cdata;
00350
00351     return epoll_ctl(epoll_fd, EPOLL_CTL_MOD, cdata->socket, ev);
00352 }
00353 }
00354
00355
00356 int construct_new_client_epoll(int epoll_fd, ClientData* cdata) {
00357
00358     struct epoll_event ev;
00359
00360     ev.events = EPOLLIN | EPOLLRDHUP | EPOLLONESHOT;
00361     ev.data.ptr = cdata;
00362
00363     return epoll_ctl(epoll_fd, EPOLL_CTL_ADD, cdata->socket, &ev);
00364 }
00365 }
00366
00367
00368 ClientData* create_new_client_data(int client_socket, Cache cache) {
00369
00370     ClientData* new_cdata = dynalloc(sizeof(ClientData), cache);
00371     if (new_cdata == NULL)
00372         return NULL;
00373
00374     memset(new_cdata->key_size_buffer, 0, LENGTH_PREFIX_SIZE);
00375     memset(new_cdata->value_size_buffer, 0, LENGTH_PREFIX_SIZE);
00376
00377     new_cdata->parsing_index = 0;
00378     new_cdata->parsing_stage = PARSING_COMMAND;
00379
00380     new_cdata->socket = client_socket;
00381
00382     new_cdata->cleaning = 0;
00383
00384     new_cdata->key = NULL;
00385     new_cdata->value = NULL;
00386
00387     return new_cdata;
00388 }
00389
00390
00391 void delete_client_data(ClientData* cdata) {
00392
00393     if (cdata == NULL)
00394         return;
00395
00396     if (cdata->socket > 0)
00397         close(cdata->socket);
00398
00399     free(cdata);
00400 }
00401 }
00402
00403
00404 void drop_client(int epoll_fd, ClientData* cdata) {
00405     epoll_ctl(epoll_fd, EPOLL_CTL_DEL, cdata->socket, NULL);
00406     delete_client_data(cdata);
00407 }
```

4.53 server/cache_server_utils.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <sys/epoll.h>
#include <netinet/in.h>
#include <wait.h>
#include <fcntl.h>
#include <errno.h>
#include <pthread.h>
#include "../dynalloc/dynalloc.h"
#include "../cache/cache.h"
#include "cache_server_models.h"
#include "../helpers/quit.h"
```

Functions

- `ssize_t recv_client (ClientData *client, char *message_buffer, int size)`
Lee hasta size bytes del client en message_buffer actualizando acordemente el indice de parseo del ClientData.
- `ssize_t clean_socket (ClientData *client, int size)`
Consume hasta size bytes del client actualizando acordemente el indice de parseo del ClientData. No guarda la informacion.
- `ssize_t send_client (ClientData *client, char *message, int size)`
Escribe size bytes del mensaje message al client objetivo.
- `int parse_request (ClientData *cdata, Cache cache)`
Parsea el mensaje proveniente del cliente asociado al data de acuerdo a su parsing stage. Este parseo siempre es total; en caso de recibirse un mensaje incompleto, la informacion del cliente se actualiza y se vuelve a invocar la funcion recursivamente hasta que se complete el mensaje o se produzca un error.
- `int handle_request (ClientData *cdata, Cache cache)`
Ejecuta el comando cargado en la estructura con informacion del cliente apuntada por cdata en su campo command. Tambien libera la memoria asignada al buffer dinamico donde se almacena la key en el cdata, en caso de ser necesario.
- `void reset_client_data (ClientData *data)`
Restablece la estructura de informacion del cliente apuntada por data, inicializando a 0 los valores asociados al stage de parseo.
- `int reconstruct_client_epoll (int epoll_fd, struct epoll_event *ev, ClientData *data)`
Reconstruye el epoll_event asociado al cliente cuya informacion se almacena en la estructura apuntada por data, y carga el evento en la instancia de epoll asociada a epoll_fd para su control.
- `int construct_new_client_epoll (int epoll_fd, ClientData *cdata)`
Reconstruye el epoll_event asociado al cliente cuya informacion se almacena en la estructura apuntada por data, y carga el evento en la instancia de epoll asociada a epoll_fd para su control.
- `ClientData * create_new_client_data (int client_socket, Cache cache)`
Crea e inicializa una nueva estructura de informacion del cliente con el socket de cliente establecido a client_↔ socket.
- `void delete_client_data (ClientData *cdata)`
Destruye la estructura de informacion del cliente objetivo.
- `void drop_client (int epoll_fd, ClientData *cdata)`
Desconecta a un cliente de la instancia epoll y destruye su estructura de cliente asociada.

4.53.1 Function Documentation

4.53.1.1 clean_socket()

```
ssize_t clean_socket (
    ClientData * client,
    int size )
```

Consume hasta `size` bytes del `client` actualizando acordemente el indice de parseo del `ClientData`. No guarda la informacion.

Parameters

out	<i>client</i>	Estructura de datos del cliente.
	<i>size</i>	Cantidad maxima de bytes a consumir.

Returns

La cantidad de bytes efectivamente consumidos, o -1 si se produjo un error.

Definition at line 9 of file `cache_server_utils.c`.

```
00009                                     {
00010
00011     int socket = client->socket;
00012
00013     ssize_t total_bytes_received = 0;
00014     ssize_t bytes_received;
00015
00016     char buffer[TRASH_BUFFER_SIZE];
00017
00018     do {
00019
00020         bytes_received = recv(socket, buffer, TRASH_BUFFER_SIZE, 0);
00021
00022         total_bytes_received += bytes_received;
00023
00024     } while (bytes_received > 0 && total_bytes_received < size);
00025
00026     if (bytes_received < 0) {
00027         perror("Error: failed to call recv bytes.");
00028         return -1;
00029     }
00030
00031     client->parsing_index += total_bytes_received;
00032
00033     return total_bytes_received;
00034 }
```

4.53.1.2 construct_new_client_epoll()

```
int construct_new_client_epoll (
    int epoll_fd,
    ClientData * cdata )
```

Reconstruye el `epoll_event` asociado al cliente cuya informacion se almacena en la estructura apuntada por `data`, y carga el evento en la instancia de `epoll` asociada a `epoll_fd` para su control.

Parameters

<i>epoll_fd</i>	File descriptor de la instancia de <code>epoll</code> objetivo.
<i>data</i>	Puntero a la estructura de informacion del cliente a controlar.

Returns

0 si la manipulacion de la instancia de `epoll` es exitosa, -1 si no.

Definition at line 356 of file `cache_server_utils.c`.

```
00356                                     {
00357
00358     struct epoll_event ev;
```

```

00359
00360     ev.events    = EPOLLIN | EPOLLRDHUP | EPOLLONESHOT;
00361     ev.data.ptr = cdata;
00362
00363     return epoll_ctl(epoll_fd, EPOLL_CTL_ADD, cdata->socket, &ev);
00364
00365 }

```

4.53.1.3 create_new_client_data()

```

ClientData * create_new_client_data (
    int client_socket,
    Cache cache )

```

Crea e inicializa una nueva estructura de informacion del cliente con el socket de cliente establecido a `client_socket`.

Parameters

<i>client_socket</i>	File descriptor del socket del cliente.
<i>cache</i>	Puntero a la cache con la que se comunicara el nuevo cliente.

Returns

Un puntero a la estructura creada e inicializada.

Definition at line 368 of file [cache_server_utils.c](#).

```

00368
00369
00370     ClientData* new_cdata = dynalloc(sizeof(ClientData), cache);
00371     if (new_cdata == NULL)
00372         return NULL;
00373
00374     memset(new_cdata->key_size_buffer, 0, LENGTH_PREFIX_SIZE);
00375     memset(new_cdata->value_size_buffer, 0, LENGTH_PREFIX_SIZE);
00376
00377     new_cdata->parsing_index = 0;
00378     new_cdata->parsing_stage = PARSING_COMMAND;
00379
00380     new_cdata->socket = client_socket;
00381
00382     new_cdata->cleaning = 0;
00383
00384     new_cdata->key = NULL;
00385     new_cdata->value = NULL;
00386
00387     return new_cdata;
00388 }

```

4.53.1.4 delete_client_data()

```

void delete_client_data (
    ClientData * cdata )

```

Destruye la estructura de informacion del cliente objetivo.

Parameters

<i>cdata</i>	El puntero a la estructura a destruir.
--------------	--

Definition at line 391 of file [cache_server_utils.c](#).

```

00391
00392
00393     if (cdata == NULL)
00394         return;
00395
00396     if (cdata->socket > 0)
00397         close(cdata->socket);
00398
00399     free(cdata);
00400
00401 }

```

4.53.1.5 drop_client()

```
void drop_client (
    int epoll_fd,
    ClientData * cdata )
```

Desconecta a un cliente de la instancia epoll y destruye su estructura de cliente asociada.

Parameters

	<i>epoll_fd</i>	File descriptor de la instancia de epoll donde se monitoreaba al cliente.
out	<i>cdata</i>	Puntero a la estructura del cliente a eliminar.

Definition at line 404 of file [cache_server_utils.c](#).

```
00404 {
00405     epoll_ctl(epoll_fd, EPOLL_CTL_DEL, cdata->socket, NULL);
00406     delete_client_data(cdata);
00407 }
```

4.53.1.6 handle_request()

```
int handle_request (
    ClientData * cdata,
    Cache cache )
```

Ejecuta el comando cargado en la estructura con informacion del cliente apuntada por *cdata* en su campo *command*. Tambien libera la memoria asignada al buffer dinamico donde se almacena la *key* en el *cdata*, en caso de ser necesario.

Parameters

	<i>cdata</i>	Puntero a la estructura con informacion del cliente.
in	<i>cache</i>	El puntero a la cache asociada al pedido a parsear.

Returns

0 si la ejecucion del pedido es exitosa, -1 si se produjo un error al intentar enviar la respuesta.

Definition at line 223 of file [cache_server_utils.c](#).

```
00223 {
00224
00225     // En todo momento, si hay un problema con el socket devolvemos -1. Del lado del server, esto hace
    que se droppee al cliente.
00226
00227     char command;
00228
00229     switch (cdata->command) {
00230
00231     case PUT:
00232
00233         int put_status = cache_put(cdata->key, cdata->key_size,
00234                                   cdata->value, cdata->value_size, cache);
00235
00236         // Si el put fue exitoso, respondemos con OK, si no, con EUNK
00237         command = put_status < 0 ? ENOTFOUND : OKAY;
00238
00239         // Si lo que se hizo fue pisar el valor, hay que liberar la memoria de la key, pues queda la
    anterior
00240         if (put_status == 1)
00241             free(cdata->key);
00242
00243         // Si se produjo un error, hay que liberar tanto la clave como el valor, pues ninguna de las dos
    fue insertada a la cache.
00244         else if (put_status < 0) {
00245             free(cdata->key);
00246             free(cdata->value);
00247         }
00248
00249         if (send_client(cdata, &command, 1) < 0) return -1;
00250
00251         break;
00252 }
```

```

00253     case DEL:
00254
00255         int del_status = cache_delete(cdata->key, cdata->key_size, cache);
00256
00257         command = del_status == 0 ? OKAY :
00258             (del_status == 1 ? ENOTFOUND : ENOTFOUND);
00259
00260         // En cualquier caso, quiero liberar el buffer donde guarde la key una vez terminada la
operacion.
00261         if (cdata->key)
00262             free(cdata->key);
00263
00264         if (send_client(cdata, &command, 1) < 0) return -1;
00265
00266         break;
00267
00268     case GET:
00269
00270         LookupResult lr = cache_get(cdata->key, cdata->key_size, cache);
00271
00272         // En cualquier caso, quiero liberar el buffer donde guarde la key una vez terminada la
operacion.
00273         if (cdata->key)
00274             free(cdata->key);
00275
00276         if (lookup_result_is_ok(lr)) {
00277
00278             char command = OKAY;
00279             char length_prefix_buffer[LENGTH_PREFIX_SIZE];
00280             size_t size = ntohl(lookup_result_get_size(lr));
00281             memcpy(length_prefix_buffer, &size, LENGTH_PREFIX_SIZE);
00282
00283             if (send_client(cdata, &command, 1) < 0) return -1;
00284             if (send_client(cdata, length_prefix_buffer, LENGTH_PREFIX_SIZE) < 0) return -1;
00285             if (send_client(cdata,
00286                 lookup_result_get_value(lr),
00287                 lookup_result_get_size(lr)) < 0) return -1;
00288         }
00289
00290         else {
00291             command = ENOTFOUND;
00292             if (send_client(cdata, &command, 1) < 0) return -1;
00293         }
00294         break;
00295
00296     case STATS: {
00297
00298         // Obtengo el reporte de estadísticas.
00299         StatsReport report = cache_report(cache);
00300
00301         // Creo un buffer donde cargar el mensaje y lo llenamos
00302         size_t buffer_size = sizeof(char) * STATS_MESSAGE_LENGTH;
00303         char report_buffer[buffer_size];
00304         char command = OKAY;
00305
00306         int report_len = stats_report_stringify(report, report_buffer);
00307
00308         char length_prefix_buffer[LENGTH_PREFIX_SIZE];
00309         size_t size = ntohl(report_len);
00310         memcpy(length_prefix_buffer, &size, LENGTH_PREFIX_SIZE);
00311
00312         // Enviamos el comando el mensaje
00313         send_client(cdata, &command, 1); // Mando STATS
00314         send_client(cdata, length_prefix_buffer, LENGTH_PREFIX_SIZE); // Prefijo longitud
00315         send_client(cdata, report_buffer, report_len); // String del report
00316
00317         break;
00318     }
00319 }
00320
00321 case EBIG:
00322
00323     command = EBIG;
00324     if (send_client(cdata, &command, 1) < 0) return -1;
00325
00326     break;
00327
00328 }
00329
00330 return 0;
00331
00332 }

```

4.53.1.7 parse_request()

```
int parse_request (
    ClientData * cdata,
    Cache cache )
```

Parsea el mensaje proveniente del cliente asociado al `data` de acuerdo a su parsing stage. Este parseo siempre es total; en caso de recibirse un mensaje incompleto, la informacion del cliente se actualiza y se vuelve a invocar la funcion recursivamente hasta que se complete el mensaje o se produzca un error.

Parameters

out	<i>cdata</i>	Puntero a la estructura con la informacion del cliente que es actualizada de acuerdo a la informacion parseada.
in	<i>cache</i>	El puntero a la cache asociada al pedido a parsear.

Returns

0 si parseo correctamente, -1 si fracaso al recibir bytes del socket.

Definition at line 95 of file `cache_server_utils.c`.

```
00095                                     {
00096
00097     switch (cdata->parsing_stage) {
00098
00099     case PARSING_COMMAND:
00100
00101         if (recv_client(cdata, &cdata->command, 1) < 0) return -1;
00102
00103         // No se termino de parsear el comando
00104         if (cdata->parsing_index < 1) return 0;
00105
00106         // Si el comando no es valido, devolvemos -1 y se le cierra la conexion
00107         if (!command_is_valid(cdata->command)) return -1;
00108
00109         // Si no, determinamos el estado de parseo dependiendo de si es STATS o no
00110         cdata->parsing_stage = cdata->command == STATS ?
00111             PARSING_FINISHED : PARSING_KEY_LEN;
00112         cdata->parsing_index = 0;
00113
00114         break;
00115
00116     case PARSING_KEY_LEN:
00117
00118         if (recv_client(cdata,
00119             cdata->key_size_buffer + cdata->parsing_index,
00120             LENGTH_PREFIX_SIZE - cdata->parsing_index) < 0) return -1;
00121
00122         if (cdata->parsing_index < LENGTH_PREFIX_SIZE) return 0;
00123
00124         cdata->key_size = htonl(*(int*)(cdata->key_size_buffer));
00125
00126         cdata->key = dynalloc(cdata->key_size, cache);
00127         if (cdata->key == NULL) {
00128             // Si falla la asignacion de memoria, marcamos el comando como EBIG.
00129             // Pasamos al cliente a estado cleaning, vaciando todo lo que queda en el buffer para poder
00130             volver a sincronizarnos con el final del mensaje.
00131             cdata->command = EBIG;
00132             cdata->cleaning = 1;
00133         }
00134
00135         cdata->parsing_stage = PARSING_KEY;
00136         cdata->parsing_index = 0;
00137
00138         break;
00139
00140     case PARSING_KEY:
00141
00142         if (cdata->cleaning) {
00143             if (clean_socket(cdata, cdata->key_size - cdata->parsing_index) < 0) {
00144                 return -1;
00145             }
00146
00147             else if (recv_client(cdata, cdata->key + cdata->parsing_index,
00148                 cdata->key_size - cdata->parsing_index) < 0) {
00149                 // Liberamos la memoria asignada hasta ahora y propagamos el error
00150                 free(cdata->key);
00151                 return -1;
```

```

00152     }
00153
00154     if (cdata->parsing_index < cdata->key_size) return 0;
00155
00156     cdata->parsing_stage = cdata->command == PUT ?
00157         PARSING_VALUE_LEN : PARSING_FINISHED;
00158     cdata->parsing_index = 0;
00159
00160     break;
00161
00162     case PARSING_VALUE_LEN:
00163
00164         if (recv_client(cdata,
00165             cdata->value_size_buffer + cdata->parsing_index,
00166             LENGTH_PREFIX_SIZE - cdata->parsing_index) < 0) {
00167             free(cdata->key);
00168             return -1;
00169         }
00170
00171         // El recv ya no puede leer mas bytes, pero no terminamos de leer el prefijo de longitud.
00172         Retornamos 0, el cliente volvera a ser controlado por el epoll, y esperamos a que vuelvan a llegar
00173         bytes para leerlos.
00174         if (cdata->parsing_index < LENGTH_PREFIX_SIZE) return 0;
00175
00176         cdata->value_size = htonl(*(int*)(cdata->value_size_buffer));
00177
00178         // Si no estaba limpiando, pido la memoria para value.
00179         if (!cdata->cleaning) {
00180
00181             cdata->value = dynalloc(cdata->value_size, cache);
00182
00183             if (cdata->value == NULL) {
00184                 // Analogo al caso de fallar en key, pero libero la memoria de key
00185                 free(cdata->key);
00186                 cdata->command = EBIG;
00187                 cdata->cleaning = 1;
00188             }
00189         }
00190
00191         cdata->parsing_stage = PARSING_VALUE;
00192         cdata->parsing_index = 0;
00193
00194         break;
00195
00196     case PARSING_VALUE:
00197
00198         if (cdata->cleaning) {
00199             if (clean_socket(cdata, cdata->key_size - cdata->parsing_index) < 0) {
00200                 return -1; // No hace falta liberar cdata->key y cdata->value nunca, porque si llegue en
00201                 cleaning a este punto ambos son NULL ya.
00202             }
00203         }
00204         else if (recv_client(cdata, cdata->value + cdata->parsing_index,
00205             cdata->value_size - cdata->parsing_index) < 0) return -1;
00206
00207         if (cdata->parsing_index < cdata->value_size) return 0;
00208
00209         cdata->parsing_stage = PARSING_FINISHED;
00210
00211         break;
00212
00213     case PARSING_FINISHED: // Imposible que llegue hasta aca, lanzamos un error
00214         quit("Error: switch case PARSING_FINISHED reached");
00215         break;
00216     }
00217
00218     if (cdata->parsing_stage != PARSING_FINISHED)
00219         return parse_request(cdata, cache);
00220
00221     return 0;

```

4.53.1.8 reconstruct_client_epoll()

```

int reconstruct_client_epoll (
    int epoll_fd,
    struct epoll_event * ev,
    ClientData * data )

```

Reconstruye el `epoll_event` asociado al cliente cuya informacion se almacena en la estructura apuntada por `data`, y carga el evento en la instancia de `epoll` asociada a `epoll_fd` para su control.

Parameters

	<i>epoll_fd</i>	File descriptor de la instancia de epoll objetivo.
out	<i>epoll_event</i>	Puntero a la estructura del epoll_event a reconstruir.
	<i>data</i>	Puntero a la estructura de informacion del cliente a controlar.

Returns

0 si la manipulacion de la instancia de epoll es exitosa, -1 si no.

Definition at line 346 of file [cache_server_utils.c](#).

```
00346                                     {
00347
00348     ev->events = EPOLLIN | EPOLLRDHUP | EPOLLONESHOT;
00349     ev->data.ptr = cdata;
00350
00351     return epoll_ctl(epoll_fd, EPOLL_CTL_MOD, cdata->socket, ev);
00352
00353 }
```

4.53.1.9 recv_client()

```
ssize_t recv_client (
    ClientData * client,
    char * message_buffer,
    int size )
```

Lee hasta size bytes del client en message_buffer actualizando acordemente el indice de parseo del ClientData.

Parameters

out	<i>client</i>	Estructura de datos del cliente.
	<i>message_buffer</i>	Buffer de lectura.
	<i>size</i>	Cantidad maxima de bytes a leer.

Returns

La cantidad de bytes efectivamente leidos, o -1 si se produjo un error.

Definition at line 37 of file [cache_server_utils.c](#).

```
00037                                     {
00038
00039     int socket = client->socket;
00040
00041     ssize_t total_bytes_received = 0;
00042     ssize_t bytes_received;
00043
00044     do {
00045
00046         bytes_received = recv(socket, message_buffer + total_bytes_received,
00047                             size - total_bytes_received, 0);
00048
00049         total_bytes_received += bytes_received;
00050
00051     } while (bytes_received > 0 && total_bytes_received < size);
00052
00053     if (bytes_received < 0) {
00054         perror("Error: failed to call recv bytes.");
00055         return -1;
00056     }
00057
00058     client->parsing_index += total_bytes_received;
00059
00060     return total_bytes_received;
00061 }
```

4.53.1.10 reset_client_data()

```
void reset_client_data (
    ClientData * data )
```

Restablece la estructura de informacion del cliente apuntada por `data`, inicializando a 0 los valores asociados al stage de parseo.

Parameters

<code>out</code>	<code>data</code>	Puntero a la estructura de informacion del cliente a restablecer.
------------------	-------------------	---

Definition at line 335 of file `cache_server_utils.c`.

```
00335 {
00336
00337     cdata->parsing_index = 0;
00338     cdata->parsing_stage = PARSING_COMMAND;
00339     cdata->cleaning      = 0;
00340     cdata->key           = NULL;
00341     cdata->value         = NULL;
00342
00343 }
```

4.53.1.11 send_client()

```
ssize_t send_client (
    ClientData * client,
    char * message,
    int size )
```

Escribe `size` bytes del mensaje `message` al `client` objetivo.

Parameters

<code>client</code>	Estructura del cliente a escribir.
<code>message</code>	Buffer con el contenido del mensaje.
<code>size</code>	Cantidad de bytes a escribir.

Returns

La cantidad de bytes enviados, que es -1 si se produjo un error.

Definition at line 64 of file `cache_server_utils.c`.

```
00064 {
00065
00066     int socket = client->socket;
00067
00068     ssize_t total_bytes_sent = 0;
00069     ssize_t bytes_sent;
00070
00071
00072     do {
00073
00074         bytes_sent = send(socket, message + total_bytes_sent,
00075                         size - total_bytes_sent, 0);
00076
00077         total_bytes_sent += bytes_sent;
00078
00079     } while (bytes_sent > 0 && total_bytes_sent < size);
00080
00081     if (bytes_sent < 0) {
00082         perror("Error: failed to call send bytes.");
00083         return -1;
00084     }
00085
00086     return total_bytes_sent;
00087
00088 }
```

4.54 cache_server_utils.h

[Go to the documentation of this file.](#)

```
00001 #ifndef __CACHE_SERVER_UTILS_H__
00002 #define __CACHE_SERVER_UTILS_H__
00003
00004 #include <stdio.h>
```



```

00005 #include <stdlib.h>
00006 #include <unistd.h>
00007 #include <string.h>
00008 #include <sys/stat.h>
00009 #include <sys/types.h>
00010 #include <sys/socket.h>
00011 #include <netinet/ip.h>
00012 #include <sys/epoll.h>
00013 #include <netinet/in.h>
00014 #include <wait.h>
00015 #include <fcntl.h>
00016 #include <errno.h>
00017 #include <pthread.h>
00018
00019 #include "../dynalloc/dynalloc.h"
00020 #include "../cache/cache.h"
00021 #include "cache_server_models.h"
00022 #include "../helpers/quit.h"
00023
00024 /**
00025  * @brief Lee hasta `size` bytes del `client` en `message_buffer` actualizando acordemente el indice
de parseo del `ClientData`.
00026  *
00027  * @param[out] client Estructura de datos del cliente.
00028  * @param message_buffer Buffer de lectura.
00029  * @param size Cantidad maxima de bytes a leer.
00030  *
00031  * @return La cantidad de bytes efectivamente leidos, o -1 si se produjo un error.
00032  */
00033 ssize_t recv_client(ClientData* client, char* message_buffer, int size);
00034
00035
00036
00037 /**
00038  * @brief Consume hasta `size` bytes del `client` actualizando acordemente el indice de parseo del
`ClientData`. No guarda la informacion.
00039  *
00040  * @param[out] client Estructura de datos del cliente.
00041  * @param size Cantidad maxima de bytes a consumir.
00042  *
00043  * @return La cantidad de bytes efectivamente consumidos, o -1 si se produjo un error.
00044  */
00045 ssize_t clean_socket(ClientData* client, int size);
00046
00047
00048
00049 /**
00050  * @brief Escribe `size` bytes del mensaje `message` al `client` objetivo.
00051  *
00052  * @param client Estructura del cliente a escribir.
00053  * @param message Buffer con el contenido del mensaje.
00054  * @param size Cantidad de bytes a escribir.
00055  *
00056  * @return La cantidad de bytes enviados, que es -1 si se produjo un error.
00057  */
00058 ssize_t send_client(ClientData* client, char* message, int size);
00059
00060 /**
00061  * @brief Parsea el mensaje proveniente del cliente asociado al `data` de acuerdo a su parsing stage.
Este parseo siempre es total; en caso de recibirse un mensaje incompleto, la informacion del cliente
se actualiza y se vuelve a invocar la funcion recursivamente hasta que se complete el mensaje o se
produza un error.
00062  *
00063  * @param[out] cdata Puntero a la estructura con la informacion del cliente que es actualizada de
acuerdo a la informacion parseada.
00064  * @param[in] cache El puntero a la cache asociada al pedido a parsear.
00065  *
00066  * @return 0 si parseo correctamente, -1 si fracaso al recibir bytes del socket.
00067  */
00068 int parse_request(ClientData* cdata, Cache cache);
00069
00070
00071 /**
00072  * @brief Ejecuta el comando cargado en la estructura con informacion del cliente apuntada por
`cdata` en su campo `command`. Tambien libera la memoria asignada al buffer dinamico donde se almacena
la `key` en el `cdata`, en caso de ser necesario.
00073  *
00074  * @param cdata Puntero a la estructura con informacion del cliente.
00075  * @param[in] cache El puntero a la cache asociada al pedido a parsear.
00076  *
00077  * @return 0 si la ejecucion del pedido es exitosa, -1 si se produjo un error al intentar enviar la
respuesta.
00078  */
00079 int handle_request(ClientData* cdata, Cache cache);
00080
00081
00082 /**

```

```

00083 * @brief Restablece la estructura de informacion del cliente apuntada por `data`, inicializando a 0
00084 * los valores asociados al stage de parseo.
00085 * @param[out] data Puntero a la estructura de informacion del cliente a restablecer.
00086 */
00087 void reset_client_data(ClientData* data);
00088
00089 /**
00090 * @brief Reconstruye el epoll_event asociado al cliente cuya informacion se almacena en la
00091 * estructura apuntada por `data`, y carga el evento en la instancia de epoll asociada a `epoll_fd` para
00092 * su control.
00093 * @param epoll_fd File descriptor de la instancia de epoll objetivo.
00094 * @param[out] epoll_event Puntero a la estructura del epoll_event a reconstruir.
00095 * @param data Puntero a la estructura de informacion del cliente a controlar.
00096 * @return 0 si la manipulacion de la instancia de epoll es exitosa, -1 si no.
00097 */
00098 int reconstruct_client_epoll(int epoll_fd, struct epoll_event* ev, ClientData* data);
00099
00100 /**
00101 * @brief Reconstruye el epoll_event asociado al cliente cuya informacion se almacena en la
00102 * estructura apuntada por `data`, y carga el evento en la instancia de epoll asociada a `epoll_fd` para
00103 * su control.
00104 * @param epoll_fd File descriptor de la instancia de epoll objetivo.
00105 * @param data Puntero a la estructura de informacion del cliente a controlar.
00106 * @return 0 si la manipulacion de la instancia de epoll es exitosa, -1 si no.
00107 */
00108 int construct_new_client_epoll(int epoll_fd, ClientData* cdata);
00109
00110 /**
00111 * @brief Crea e inicializa una nueva estructura de informacion del cliente con el socket de cliente
00112 * establecido a `client_socket`.
00113 * @param client_socket File descriptor del socket del cliente.
00114 * @param cache Puntero a la cache con la que se comunicara el nuevo cliente.
00115 * @return Un puntero a la estructura creada e inicializada.
00116 */
00117 ClientData* create_new_client_data(int client_socket, Cache cache);
00118
00119 /**
00120 * @brief Destruye la estructura de informacion del cliente objetivo.
00121 * @param cdata El puntero a la estructura a destruir.
00122 */
00123 void delete_client_data(ClientData* cdata);
00124
00125 /**
00126 * @brief Desconecta a un cliente de la instancia epoll y destruye su estructura de cliente asociada.
00127 * @param epoll_fd File descriptor de la instancia de epoll donde se monitoreaba al cliente.
00128 * @param[out] cdata Puntero a la estructura del cliente a eliminar.
00129 */
00130 void drop_client(int epoll_fd, ClientData* cdata);
00131
00132 #endif // __CACHE_SERVER_UTILS_H__

```

4.55 server/server_starter.c File Reference

```
#include "server_starter_utils.h"
```

Functions

- int [main](#) (int argc, char **argv)

Lee los argumentos de entrada como numero de puerto, cantidad de memoria, y numero de threads, e inicializa el socket del servidor MemCached, que es pasado por argumento al servidor que ejecuta inmediatamente despues.

4.55.1 Function Documentation

4.55.1.1 main()

```
int main (
    int argc,
    char ** argv )
```

Lee los argumentos de entrada como numero de puerto, cantidad de memoria, y numero de threads, e inicializa el socket del servidor MemCached, que es pasado por argumento al servidor que ejecuta inmediatamente despues.

Uso:

`[-p port] o [--port port]` para especificar el puerto que escuchara el server.

`[-m mem] o [--memory mem]` para especificar la cantidad de memoria maxima en bytes.

`[-t num] o [--threads num]` para especificar la cantidad de threads a lanzar.

Returns

0 en caso de ejecutar correctamente, distinto de 0 si no.

Definition at line 18 of file [server_starter.c](#).

```
00018 {
00019
00020     Args args;
00021
00022     if (parse_arguments(argc, argv, &args))
00023         return 1;
00024
00025     int server_socket = create_server_socket(args.port);
00026
00027     set_memory_limit(args.memory_limit);
00028
00029     printf("[Port]      %d\n", args.port);
00030     printf("[Memory]    %ld Bytes\n", args.memory_limit);
00031     printf("[Threads]  %d\n", args.num_threads);
00032
00033     exec_server("./bin/cache_server", server_socket, args.num_threads);
00034
00035     return 0;
00036 }
```

4.56 server_starter.c

[Go to the documentation of this file.](#)

```
00001 #include "server_starter_utils.h"
00002
00003 /**
00004  * @brief Lee los argumentos de entrada como numero de puerto, cantidad de memoria, y numero de
00005  * threads, e inicializa el socket del servidor MemCached, que es pasado por argumento al servidor que
00006  * ejecuta inmediatamente despues.
00007  *
00008  * `Uso`:
00009  *
00010  *     [-p port] o [--port port] para especificar el puerto que escuchara el server.
00011  *
00012  *     [-m mem] o [--memory mem] para especificar la cantidad de memoria maxima en bytes.
00013  *
00014  *     [-t num] o [--threads num] para especificar la cantidad de threads a lanzar.
00015  *
00016  * @return 0 en caso de ejecutar correctamente, distinto de 0 si no.
00017  */
00018 int main(int argc, char** argv) {
00019
00020     Args args;
00021
00022     if (parse_arguments(argc, argv, &args))
00023         return 1;
00024
00025     int server_socket = create_server_socket(args.port);
00026
00027     set_memory_limit(args.memory_limit);
00028
00029     printf("[Port]      %d\n", args.port);
00030     printf("[Memory]    %ld Bytes\n", args.memory_limit);
```

```

00031  printf("[Threads] %d\n", args.num_threads);
00032
00033  exec_server("./bin/cache_server", server_socket, args.num_threads);
00034
00035  return 0;
00036 }

```

4.57 server/server_starter_utils.c File Reference

```

#include <string.h>
#include <ctype.h>
#include "server_starter_utils.h"
#include "../helpers/quit.h"

```

Macros

- #define [BACKLOG_SIZE](#) 100
- #define [DEFAULT_PORT](#) 889

Functions

- int [parse_arguments](#) (int argc, char **argv, [Args](#) *args)
*Recibe el `argc` y `argv` de `main` y parsea sus contenidos cargandolos en `*args`. Reconoce las banderas definidas en `main` y establece valores defaults en caso de omitirse alguna.*
- int [create_server_socket](#) (int port)
Inicializa el socket donde el servidor escuchara las nuevas conexiones. El socket se bindea al `port` TCP pasado por argumento, con `domain AF_INET` y tipo `stream`, y se coloca en modo escucha.
- void [set_memory_limit](#) (unsigned long memory_limit)
Define el tope de memoria disponible para el proceso llamante, setteando tanto el `soft` como el `hard limit` a `memory↔_limit`, medido en bytes.
- void [exec_server](#) (char *program, int socket, int threads)
Ejecuta el servidor cuyo ejecutable se encuentra en el path `program`, pasandole como argumentos el file descriptor de su socket asociado `socket` y la cantidad de threads `threads`.

4.57.1 Macro Definition Documentation

4.57.1.1 BACKLOG_SIZE

```
#define BACKLOG_SIZE 100
```

Definition at line 7 of file [server_starter_utils.c](#).

4.57.1.2 DEFAULT_PORT

```
#define DEFAULT_PORT 889
```

Definition at line 8 of file [server_starter_utils.c](#).

4.57.2 Function Documentation

4.57.2.1 create_server_socket()

```

int create_server_socket (
    int port )

```

Inicializa el socket donde el servidor escuchara las nuevas conexiones. El socket se bindea al `port` TCP pasado por argumento, con `domain AF_INET` y tipo `stream`, y se coloca en modo escucha.

Parameters

<code>in</code>	<code>port</code>	Numero de puerto TCP al que se bindeara el socket.
-----------------	-------------------	--

Returns

El file descriptor asociado al socket creado.

Definition at line 109 of file [server_starter_utils.c](#).

```

00109     {
00110
00111     int server_socket = socket(AF_INET, SOCK_STREAM, 0);
00112     if (server_socket < 0)
00113         quit("Error: failed to create socket");
00114
00115     int yes = 1;
00116     if (setsockopt(server_socket, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof yes) < 0)
00117         quit("Error: failed to set socket options");
00118
00119     struct sockaddr_in server_saddr;
00120     server_saddr.sin_family = AF_INET;
00121     server_saddr.sin_port = htons(port);
00122     server_saddr.sin_addr.s_addr = htonl(INADDR_ANY);
00123
00124     // Lo bindemos al puerto que nos pasaron
00125     if (bind(server_socket, (struct sockaddr*) &server_saddr, sizeof server_saddr) < 0)
00126         quit("Error: failed to bind socket.");
00127
00128     // Ponemos al socket en modo escucha
00129     if (listen(server_socket, BACKLOG_SIZE) < 0)
00130         quit("Error: failed to set the socket to listen mode");
00131
00132     return server_socket;
00133 }
```

4.57.2.2 exec_server()

```

void exec_server (
    char * program,
    int socket,
    int threads )
```

Ejecuta el servidor cuyo ejecutable se encuentra en el path `program`, pasandole como argumentos el file descriptor de su socket asociado `socket` y la cantidad de threads `threads`.

Parameters

<i>program</i>	El path al ejecutable del servidor memcached.
<i>socket</i>	El file descriptor del socket asociado al servidor.
<i>threads</i>	La cantidad de threads con las que se lanzara el server.

Definition at line 145 of file [server_starter_utils.c](#).

```

00145     {
00146
00147     char socket_buffer[100];
00148     char threads_buffer[100];
00149     sprintf(socket_buffer, "%d", socket);
00150     sprintf(threads_buffer, "%d", threads);
00151
00152     if (execl(program, program, socket_buffer, threads_buffer, NULL) < 0)
00153         quit("Error: failed to exec the cache server.");
00154 }
```

4.57.2.3 parse_arguments()

```

int parse_arguments (
    int argc,
    char ** argv,
    Args * args )
```

Recibe el `argc` y `argv` de `main` y parsea sus contenidos cargandolos en `*args`. Reconoce las banderas definidas en `main` y establece valores defaults en caso de omitirse alguna.

Parameters

in	<i>argc</i>	Cantidad de argumentos leidos al ejecutarse el programa.
in	<i>argv</i>	Array de argumentos leidos.

Parameters

out	args	Puntero a la estructura Args donde cargaremos los resultados del parseo.
-----	------	--

Returns

0 en caso de exito, 1 si el usuario solicito ayuda para la ejecucion.

Definition at line 49 of file [server_starter_utils.c](#).

```

00049                                     {
00050
00051     // Chequeamos si el usuario solicito ayuda para el uso del programa
00052     if (check_for_usage(argc, argv)) {
00053         show_usage();
00054         return 1;
00055     }
00056
00057     // Definimos los valores predeterminados del server
00058     long pages = sysconf(_SC_PHYS_PAGES);
00059     long page_size = sysconf(_SC_PAGE_SIZE);
00060     unsigned long total_ram = pages * page_size;
00061
00062     args->memory_limit = total_ram;
00063     args->num_threads = sysconf(_SC_NPROCESSORS_ONLN);
00064     args->port = DEFAULT_PORT;
00065
00066     // Parseamos buscando cada posible bandera y realizando chequeos
00067     for (int i = 1 ; i < argc ; i++) {
00068
00069         if (strcmp("-p", argv[i]) == 0 || strcmp("--port", argv[i]) == 0) {
00070
00071             if (i + 1 < argc && is_positive_integer(argv[i+1])) {
00072                 args->port = atoi(argv[i+1]);
00073                 i++;
00074             }
00075
00076             else return printf("server: error: not a valid port\n");
00077         }
00078
00079         else if (strcmp("-m", argv[i]) == 0 || strcmp("--memory", argv[i]) == 0) {
00080
00081             if (i + 1 < argc && is_positive_integer(argv[i+1])) {
00082
00083                 args->memory_limit = strtoul(argv[i+1], NULL, 10);
00084                 i++;
00085             }
00086
00087             else return printf("server: error: not a valid memory limit\n");
00088         }
00089
00090         else if (strcmp("-t", argv[i]) == 0 || strcmp("--threads", argv[i]) == 0) {
00091
00092             if (i + 1 < argc && is_positive_integer(argv[i+1])) {
00093                 args->num_threads = atoi(argv[i+1]);
00094                 i++;
00095             }
00096
00097             else return printf("server: error: not a valid number of threads\n");
00098         }
00099
00100         else return printf("server: invalid option: %s\n", argv[i]);
00101     }
00102
00103     return 0;
00104 }
00105
00106 }
```

4.57.2.4 set_memory_limit()

```

void set_memory_limit (
    unsigned long memory_limit )
```

Define el tope de memoria disponible para el proceso llamante, seteando tanto el soft como el hard limit a memory_limit, medido en bytes.

Parameters

memory_limit	Cantidad de memoria maxima medida en bytes.
--------------	---

Definition at line 136 of file [server_starter_utils.c](#).

```
00136                                     {
00137
00138     struct rlimit rlim = {memory_limit, memory_limit};
00139
00140     if (setrlimit(RLIMIT_DATA, &rlim) < 0)
00141         quit("Error: failed to set memory limit.");
00142 }
```

4.58 server_starter_utils.c

[Go to the documentation of this file.](#)

```
00001 #include <string.h>
00002 #include <ctype.h>
00003
00004 #include "server_starter_utils.h"
00005 #include "../helpers/quit.h"
00006
00007 #define BACKLOG_SIZE 100
00008 #define DEFAULT_PORT 889
00009
00010 /**
00011  * @brief Imprime en pantalla la forma de ejecutar el server, indicando sus banderas posibles.
00012  */
00013 static void show_usage() {
00014
00015     printf("Usage: ./server [options]\n");
00016     printf("Options:\n");
00017     printf("  -p, --port <port_number>    Set the port number in which the server will run.\n");
00018     printf("  -m, --memory <memory_limit> Set the memory limit of the server.\n");
00019     printf("  -t, --threads <num_threads> Set the number of threads of the server.\n");
00020 }
00021
00022 /**
00023  * @brief Chequea si el usuario ha solicitado ayuda para ejecutar el servidor mediante la bandera
00024  *        '--help' o '-h'.
00025  * @param [in] argc Arreglo Cantidad de argumentos leidos al ejecutarse el programa.
00026  * @param [in] argv Array de argumentos leidos.
00027  * @return Devuelve 1 si se solicito ayuda y 0 en caso contrario.
00028  */
00029 static int check_for_usage(int argc, char** argv) {
00030
00031     for (int i = 0 ; i < argc ; i++)
00032         if (strcmp(argv[i], "--help") == 0 || strcmp(argv[i], "-h") == 0) return 1;
00033
00034     return 0;
00035 }
00036
00037 /**
00038  * @brief Determina si la cadena ingresada se corresponde con un numero entero positivo.
00039  * @param [in] s Cadena para la cual queremos analizar si su contenido es un entero positivo.
00040  * @return 1 si es un entero positivo, 0 si no.
00041  */
00042 static int is_positive_integer(char* s) {
00043
00044     for (int i = 0 ; s[i] != '\0' ; i++)
00045         if (!isdigit(s[i])) return 0;
00046
00047     return 1;
00048 }
00049 int parse_arguments(int argc, char** argv, Args* args) {
00050
00051     // Chequeamos si el usuario solicito ayuda para el uso del programa
00052     if (check_for_usage(argc, argv)) {
00053         show_usage();
00054         return 1;
00055     }
00056
00057     // Definimos los valores predeterminados del server
00058     long pages = sysconf(_SC_PHYS_PAGES);
00059     long page_size = sysconf(_SC_PAGE_SIZE);
00060     unsigned long total_ram = pages * page_size;
00061
00062     args->memory_limit = total_ram;
00063     args->num_threads = sysconf(_SC_NPROCESSORS_ONLN);
00064     args->port = DEFAULT_PORT;
00065
00066     // Parseamos buscando cada posible bandera y realizando chequeos
00067     for (int i = 1 ; i < argc ; i++) {
00068
00069         if (strcmp("-p", argv[i]) == 0 || strcmp("--port", argv[i]) == 0) {
00070
00071             if (i + 1 < argc && is_positive_integer(argv[i+1])) {
```

```

00072         args->port = atoi(argv[i+1]);
00073         i++;
00074     }
00075
00076     else return printf("server: error: not a valid port\n");
00077 }
00078
00079     else if (strcmp("-m", argv[i]) == 0 || strcmp("--memory", argv[i]) == 0) {
00080
00081         if (i + 1 < argc && is_positive_integer(argv[i+1])) {
00082
00083             args->memory_limit = strtoul(argv[i+1], NULL, 10);
00084             i++;
00085
00086         }
00087
00088         else return printf("server: error: not a valid memory limit\n");
00089     }
00090
00091     else if (strcmp("-t", argv[i]) == 0 || strcmp("--threads", argv[i]) == 0) {
00092
00093         if (i + 1 < argc && is_positive_integer(argv[i+1])) {
00094             args->num_threads = atoi(argv[i+1]);
00095             i++;
00096         }
00097
00098         else return printf("server: error: not a valid number of threads\n");
00099     }
00100 }
00101
00102     else return printf("server: invalid option: %s\n", argv[i]);
00103 }
00104
00105 return 0;
00106 }
00107
00108
00109 int create_server_socket(int port) {
00110
00111     int server_socket = socket(AF_INET, SOCK_STREAM, 0);
00112     if (server_socket < 0)
00113         quit("Error: failed to create socket");
00114
00115     int yes = 1;
00116     if (setsockopt(server_socket, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof yes) < 0)
00117         quit("Error: failed to set socket options");
00118
00119     struct sockaddr_in server_saddr;
00120     server_saddr.sin_family = AF_INET;
00121     server_saddr.sin_port = htons(port);
00122     server_saddr.sin_addr.s_addr = htonl(INADDR_ANY);
00123
00124     // Lo bindemos al puerto que nos pasaron
00125     if (bind(server_socket, (struct sockaddr*)&server_saddr, sizeof server_saddr) < 0)
00126         quit("Error: failed to bind socket.");
00127
00128     // Ponemos al socket en modo escucha
00129     if (listen(server_socket, BACKLOG_SIZE) < 0)
00130         quit("Error: failed to set the socket to listen mode");
00131
00132     return server_socket;
00133 }
00134
00135
00136 void set_memory_limit(unsigned long memory_limit){
00137
00138     struct rlimit rlim = {memory_limit, memory_limit};
00139
00140     if (setrlimit(RLIMIT_DATA, &rlim) < 0)
00141         quit("Error: failed to set memory limit.");
00142 }
00143
00144
00145 void exec_server(char* program, int socket, int threads) {
00146
00147     char socket_buffer[100];
00148     char threads_buffer[100];
00149     sprintf(socket_buffer, "%d", socket);
00150     sprintf(threads_buffer, "%d", threads);
00151
00152     if (execl(program, program, socket_buffer, threads_buffer, NULL) < 0)
00153         quit("Error: failed to exec the cache server.");
00154 }

```


4.59 server/server_starter_utils.h File Reference

```
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/ip.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/epoll.h>
#include <netinet/in.h>
#include <wait.h>
#include <fcntl.h>
#include <stdlib.h>
#include <ctype.h>
```

Classes

- struct [Args](#)

Macros

- #define [DEFAULT_PORT](#) 889

Functions

- int [parse_arguments](#) (int argc, char **argv, [Args](#) *args)
*Recibe el `argc` y `argv` de `main` y parsea sus contenidos cargandolos en `*args`. Reconoce las banderas definidas en `main` y establece valores defaults en caso de omitirse alguna.*
- int [create_server_socket](#) (int port)
Inicializa el socket donde el servidor escuchara las nuevas conexiones. El socket se bindea al `port` TCP pasado por argumento, con `domain AF_INET` y tipo `stream`, y se coloca en modo escucha.
- void [set_memory_limit](#) (unsigned long memory_limit)
Define el tope de memoria disponible para el proceso llamante, setteando tanto el soft como el hard limit a `memory_limit`, medido en bytes.
- void [exec_server](#) (char *program, int socket, int threads)
Ejecuta el servidor cuyo ejecutable se encuentra en el path `program`, pasandole como argumentos el file descriptor de su socket asociado `socket` y la cantidad de threads `threads`.

4.59.1 Macro Definition Documentation

4.59.1.1 DEFAULT_PORT

```
#define DEFAULT_PORT 889
```

Definition at line 29 of file [server_starter_utils.h](#).

4.59.2 Function Documentation

4.59.2.1 create_server_socket()

```
int create_server_socket (
    int port )
```

Inicializa el socket donde el servidor escuchara las nuevas conexiones. El socket se bindea al `port` TCP pasado por argumento, con `domain AF_INET` y tipo `stream`, y se coloca en modo escucha.

Parameters

<code>in</code>	<code>port</code>	Numero de puerto TCP al que se bindeara el socket.
-----------------	-------------------	--

Returns

El file descriptor asociado al socket creado.

Definition at line 109 of file [server_starter_utils.c](#).

```

00109     {
00110
00111     int server_socket = socket(AF_INET, SOCK_STREAM, 0);
00112     if (server_socket < 0)
00113         quit("Error: failed to create socket");
00114
00115     int yes = 1;
00116     if (setsockopt(server_socket, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof yes) < 0)
00117         quit("Error: failed to set socket options");
00118
00119     struct sockaddr_in server_saddr;
00120     server_saddr.sin_family = AF_INET;
00121     server_saddr.sin_port = htons(port);
00122     server_saddr.sin_addr.s_addr = htonl(INADDR_ANY);
00123
00124     // Lo bindemos al puerto que nos pasaron
00125     if (bind(server_socket, (struct sockaddr*) &server_saddr, sizeof server_saddr) < 0)
00126         quit("Error: failed to bind socket.");
00127
00128     // Ponemos al socket en modo escucha
00129     if (listen(server_socket, BACKLOG_SIZE) < 0)
00130         quit("Error: failed to set the socket to listen mode");
00131
00132     return server_socket;
00133 }
```

4.59.2.2 exec_server()

```

void exec_server (
    char * program,
    int socket,
    int threads )
```

Ejecuta el servidor cuyo ejecutable se encuentra en el path `program`, pasandole como argumentos el file descriptor de su socket asociado `socket` y la cantidad de threads `threads`.

Parameters

<i>program</i>	El path al ejecutable del servidor memcached.
<i>socket</i>	El file descriptor del socket asociado al servidor.
<i>threads</i>	La cantidad de threads con las que se lanzara el server.

Definition at line 145 of file [server_starter_utils.c](#).

```

00145     {
00146
00147     char socket_buffer[100];
00148     char threads_buffer[100];
00149     sprintf(socket_buffer, "%d", socket);
00150     sprintf(threads_buffer, "%d", threads);
00151
00152     if (execl(program, program, socket_buffer, threads_buffer, NULL) < 0)
00153         quit("Error: failed to exec the cache server.");
00154 }
```

4.59.2.3 parse_arguments()

```

int parse_arguments (
    int argc,
    char ** argv,
    Args * args )
```

Recibe el `argc` y `argv` de `main` y parsea sus contenidos cargandolos en `*args`. Reconoce las banderas definidas en `main` y establece valores defaults en caso de omitirse alguna.

Parameters

in	<i>argc</i>	Cantidad de argumentos leídos al ejecutarse el programa.
in	<i>argv</i>	Array de argumentos leídos.

Parameters

out	args	Puntero a la estructura Args donde cargaremos los resultados del parseo.
-----	------	--

Returns

0 en caso de exito, 1 si el usuario solicito ayuda para la ejecucion.

Definition at line 49 of file [server_starter_utils.c](#).

```

00049                                     {
00050
00051     // Chequeamos si el usuario solicito ayuda para el uso del programa
00052     if (check_for_usage(argc, argv)) {
00053         show_usage();
00054         return 1;
00055     }
00056
00057     // Definimos los valores predeterminados del server
00058     long pages = sysconf(_SC_PHYS_PAGES);
00059     long page_size = sysconf(_SC_PAGE_SIZE);
00060     unsigned long total_ram = pages * page_size;
00061
00062     args->memory_limit = total_ram;
00063     args->num_threads = sysconf(_SC_NPROCESSORS_ONLN);
00064     args->port = DEFAULT_PORT;
00065
00066     // Parseamos buscando cada posible bandera y realizando chequeos
00067     for (int i = 1 ; i < argc ; i++) {
00068
00069         if (strcmp("-p", argv[i]) == 0 || strcmp("--port", argv[i]) == 0) {
00070
00071             if (i + 1 < argc && is_positive_integer(argv[i+1])) {
00072                 args->port = atoi(argv[i+1]);
00073                 i++;
00074             }
00075
00076             else return printf("server: error: not a valid port\n");
00077         }
00078
00079         else if (strcmp("-m", argv[i]) == 0 || strcmp("--memory", argv[i]) == 0) {
00080
00081             if (i + 1 < argc && is_positive_integer(argv[i+1])) {
00082
00083                 args->memory_limit = strtoul(argv[i+1], NULL, 10);
00084                 i++;
00085             }
00086
00087             else return printf("server: error: not a valid memory limit\n");
00088         }
00089
00090         else if (strcmp("-t", argv[i]) == 0 || strcmp("--threads", argv[i]) == 0) {
00091
00092             if (i + 1 < argc && is_positive_integer(argv[i+1])) {
00093                 args->num_threads = atoi(argv[i+1]);
00094                 i++;
00095             }
00096
00097             else return printf("server: error: not a valid number of threads\n");
00098         }
00099
00100         else return printf("server: invalid option: %s\n", argv[i]);
00101     }
00102
00103     return 0;
00104 }
00105
00106 }
```

4.59.2.4 set_memory_limit()

```

void set_memory_limit (
    unsigned long memory_limit )
```

Define el tope de memoria disponible para el proceso llamante, seteando tanto el soft como el hard limit a `memory_limit`, medido en bytes.

Parameters

<i>memory_limit</i>	Cantidad de memoria maxima medida en bytes.
---------------------	---

Definition at line 136 of file [server_starter_utils.c](#).

```
00136                                     {
00137
00138     struct rlimit rlim = {memory_limit, memory_limit};
00139
00140     if (setrlimit(RLIMIT_DATA, &rlim) < 0)
00141         quit("Error: failed to set memory limit.");
00142 }
```

4.60 server_starter_utils.h

[Go to the documentation of this file.](#)

```
00001 #ifndef __SERVER_STARTER_UTILS_H__
00002 #define __SERVER_STARTER_UTILS_H__
00003
00004 #include <stdio.h>
00005 #include <unistd.h>
00006 #include <sys/socket.h>
00007 #include <string.h>
00008 #include <sys/types.h>
00009 #include <netinet/ip.h>
00010 #include <sys/time.h>
00011 #include <sys/resource.h>
00012 #include <sys/epoll.h>
00013 #include <netinet/in.h>
00014 #include <wait.h>
00015 #include <fcntl.h>
00016 #include <stdlib.h>
00017 #include <string.h>
00018 #include <ctype.h>
00019
00020
00021 typedef struct {
00022
00023     int port;
00024     unsigned long memory_limit;
00025     int num_threads;
00026
00027 } Args;
00028
00029 #define DEFAULT_PORT 889
00030
00031 /**
00032  * @brief Recibe el `argc` y `argv` de `main` y parsea sus contenidos cargandolos en `*args`.
00033  * Reconoce las banderas definidas en main y establece valores defaults en caso de omitirse alguna.
00034  *
00035  * @param[in] argc Cantidad de argumentos leidos al ejecutarse el programa.
00036  * @param[in] argv Array de argumentos leidos.
00037  * @param[out] args Puntero a la estructura Args donde cargaremos los resultados del parseo.
00038  *
00039  * @return 0 en caso de exito, 1 si el usuario solicito ayuda para la ejecucion.
00040  */
00041 int parse_arguments(int argc, char** argv, Args* args);
00042
00043 /**
00044  * @brief Inicializa el socket donde el servidor escuchara las nuevas conexiones. El socket se bindea
00045  * al `port` TCP pasado por argumento, con `domain` `AF_INET` y tipo stream, y se coloca en modo escucha.
00046  *
00047  * @param[in] port Numero de puerto TCP al que se bindeara el socket.
00048  *
00049  * @return El file descriptor asociado al socket creado.
00050  */
00051 int create_server_socket(int port);
00052
00053 /**
00054  * @brief Define el tope de memoria disponible para el proceso llamante, setteando tanto el soft como
00055  * el hard limit a `memory_limit`, medido en bytes.
00056  *
00057  * @param memory_limit Cantidad de memoria maxima medida en bytes.
00058  */
00059 void set_memory_limit(unsigned long memory_limit);
00060
00061 /**
00062  * @brief Ejecuta el servidor cuyo ejecutable se encuentra en el path `program`, pasandole como
00063  * argumentos el file descriptor de su socket asociado `socket` y la cantidad de threads `threads`.
00064  *
00065  * @param program El path al ejecutable del servidor memcached.
00066  * @param socket El file descriptor del socket asociado al servidor.
00067  * @param threads La cantidad de threads con las que se lanzara el server.
00068  */
00069 void exec_server(char* program, int socket, int threads);
```

```
00069
00070
00071 #endif // __SERVER_STARTER_UTILS_H__
```


Index

- allocated_memory
 - CacheStats, 8
 - StatsReport, 16
- app/main.c, 19, 23
- app/main2.c, 25, 29
- app/main3.c, 31, 34
- Args, 5
 - memory_limit, 5
 - num_threads, 5
 - port, 5
- atom_counter.c
 - atom_counter_add, 116
 - atom_counter_create, 116
 - atom_counter_dec, 116
 - atom_counter_destroy, 116
 - atom_counter_drop, 117
 - atom_counter_get, 117
 - atom_counter_inc, 117
- atom_counter.h
 - atom_counter_add, 120
 - atom_counter_create, 120
 - atom_counter_dec, 121
 - atom_counter_destroy, 121
 - atom_counter_drop, 121
 - atom_counter_get, 122
 - atom_counter_inc, 122
 - AtomCounter, 120
 - Counter, 120
 - COUNTER_FORMAT, 120
- atom_counter_add
 - atom_counter.c, 116
 - atom_counter.h, 120
- atom_counter_create
 - atom_counter.c, 116
 - atom_counter.h, 120
- atom_counter_dec
 - atom_counter.c, 116
 - atom_counter.h, 121
- atom_counter_destroy
 - atom_counter.c, 116
 - atom_counter.h, 121
- atom_counter_drop
 - atom_counter.c, 117
 - atom_counter.h, 121
- atom_counter_get
 - atom_counter.c, 117
 - atom_counter.h, 122
- atom_counter_inc
 - atom_counter.c, 117
 - atom_counter.h, 122
- atom_counter.h, 122
- AtomCounter, 6
 - atom_counter.h, 120
 - counter, 6
 - lock, 6
- BACKLOG_SIZE
 - server_starter_utils.c, 190
- BLUE
 - cache_server.c, 155
- bucket_number
 - LRUNode, 14
- buckets
 - Cache, 7
- BUCKETS_FACTOR
 - cache.c, 36
- Cache, 6
 - buckets, 7
 - cache.h, 50
 - dynalloc.h, 86
 - hash_function, 7
 - lrunode.h, 150
 - num_buckets, 7
 - num_zones, 7
 - queue, 7
 - stats, 7
 - zone_locks, 7
- cache
 - ServerArgs, 15
 - ThreadArgs, 18
- cache.c
 - BUCKETS_FACTOR, 36
 - cache_create, 37
 - cache_delete, 37
 - cache_destroy, 38
 - cache_free_up_memory, 39
 - cache_get, 40
 - cache_get_cstats, 41
 - cache_put, 41
 - cache_report, 43
 - ZONES_FACTOR, 36
- cache.h
 - Cache, 50
 - cache_create, 51
 - cache_delete, 51
 - cache_destroy, 52
 - cache_free_up_memory, 53
 - cache_get, 54
 - cache_get_cstats, 55

- cache_put, 55
- cache_report, 57
- DEBUG, 50
- HashFunction, 50
- kr_hash, 57
- PRINT, 50
- cache/cache.c, 35, 43
- cache/cache.h, 49, 58
- cache/cache_stats.c, 59, 68
- cache/cache_stats.h, 70, 80
- cache_create
 - cache.c, 37
 - cache.h, 51
- cache_delete
 - cache.c, 37
 - cache.h, 51
- cache_destroy
 - cache.c, 38
 - cache.h, 52
- cache_free_up_memory
 - cache.c, 39
 - cache.h, 53
- cache_get
 - cache.c, 40
 - cache.h, 54
- cache_get_cstats
 - cache.c, 41
 - cache.h, 55
- cache_put
 - cache.c, 41
 - cache.h, 55
- cache_report
 - cache.c, 43
 - cache.h, 57
- cache_server.c
 - BLUE, 155
 - GREEN, 155
 - main, 156
 - ORANGE, 155
 - RED, 155
 - RESET, 155
 - SOFT_RED, 156
 - start_server, 156
 - working_thread, 157
- cache_server_models.h
 - Command, 162
 - DEL, 162
 - EBIG, 163
 - EBINARY, 162
 - EINVALID, 162
 - ENOTFOUND, 162
 - EUNK, 163
 - GET, 162
 - LENGTH_PREFIX_SIZE, 162
 - OKAY, 162
 - PARSING_COMMAND, 162
 - PARSING_FINISHED, 162
 - PARSING_KEY, 162
 - PARSING_KEY_LEN, 162
 - PARSING_VALUE, 162
 - PARSING_VALUE_LEN, 162
 - ParsingStage, 162
 - PUT, 162
 - Response, 162
 - STATS, 162
- cache_server_utils.c
 - clean_socket, 165
 - construct_new_client_epoll, 165
 - create_new_client_data, 166
 - delete_client_data, 166
 - drop_client, 167
 - handle_request, 167
 - parse_request, 169
 - reconstruct_client_epoll, 171
 - recv_client, 171
 - reset_client_data, 172
 - send_client, 172
 - TRASH_BUFFER_SIZE, 165
- cache_server_utils.h
 - clean_socket, 179
 - construct_new_client_epoll, 179
 - create_new_client_data, 180
 - delete_client_data, 180
 - drop_client, 180
 - handle_request, 181
 - parse_request, 182
 - reconstruct_client_epoll, 184
 - recv_client, 185
 - reset_client_data, 185
 - send_client, 186
- cache_stats.c
 - cache_stats_allocated_memory_add, 60
 - cache_stats_allocated_memory_free, 61
 - cache_stats_create, 61
 - cache_stats_del_counter_dec, 61
 - cache_stats_del_counter_inc, 62
 - cache_stats_destroy, 62
 - cache_stats_evict_counter_dec, 63
 - cache_stats_evict_counter_inc, 63
 - cache_stats_get_allocated_memory, 64
 - cache_stats_get_counter_dec, 64
 - cache_stats_get_counter_inc, 64
 - cache_stats_key_counter_dec, 65
 - cache_stats_key_counter_inc, 65
 - cache_stats_put_counter_dec, 66
 - cache_stats_put_counter_inc, 66
 - cache_stats_report, 66
 - cache_stats_show, 67
 - stats_report_stringify, 67
- cache_stats.h
 - cache_stats_allocated_memory_add, 72
 - cache_stats_allocated_memory_free, 72
 - cache_stats_create, 72
 - cache_stats_del_counter_dec, 73
 - cache_stats_del_counter_inc, 73
 - cache_stats_destroy, 74

- cache_stats_evict_counter_dec, [74](#)
- cache_stats_evict_counter_inc, [75](#)
- cache_stats_get_allocated_memory, [75](#)
- cache_stats_get_counter_dec, [75](#)
- cache_stats_get_counter_inc, [76](#)
- cache_stats_key_counter_dec, [76](#)
- cache_stats_key_counter_inc, [77](#)
- cache_stats_put_counter_dec, [77](#)
- cache_stats_put_counter_inc, [77](#)
- cache_stats_report, [79](#)
- cache_stats_show, [79](#)
- CacheStats, [71](#)
- STATS_COUNT, [71](#)
- STATS_MESSAGE_LENGTH, [71](#)
- stats_report_stringify, [80](#)
- StatsReport, [71](#)
- cache_stats_allocated_memory_add
 - cache_stats.c, [60](#)
 - cache_stats.h, [72](#)
- cache_stats_allocated_memory_free
 - cache_stats.c, [61](#)
 - cache_stats.h, [72](#)
- cache_stats_create
 - cache_stats.c, [61](#)
 - cache_stats.h, [72](#)
- cache_stats_del_counter_dec
 - cache_stats.c, [61](#)
 - cache_stats.h, [73](#)
- cache_stats_del_counter_inc
 - cache_stats.c, [62](#)
 - cache_stats.h, [73](#)
- cache_stats_destroy
 - cache_stats.c, [62](#)
 - cache_stats.h, [74](#)
- cache_stats_evict_counter_dec
 - cache_stats.c, [63](#)
 - cache_stats.h, [74](#)
- cache_stats_evict_counter_inc
 - cache_stats.c, [63](#)
 - cache_stats.h, [75](#)
- cache_stats_get_allocated_memory
 - cache_stats.c, [64](#)
 - cache_stats.h, [75](#)
- cache_stats_get_counter_dec
 - cache_stats.c, [64](#)
 - cache_stats.h, [75](#)
- cache_stats_get_counter_inc
 - cache_stats.c, [64](#)
 - cache_stats.h, [76](#)
- cache_stats_key_counter_dec
 - cache_stats.c, [65](#)
 - cache_stats.h, [76](#)
- cache_stats_key_counter_inc
 - cache_stats.c, [65](#)
 - cache_stats.h, [77](#)
- cache_stats_put_counter_dec
 - cache_stats.c, [66](#)
 - cache_stats.h, [77](#)
- cache_stats_put_counter_inc
 - cache_stats.c, [66](#)
 - cache_stats.h, [77](#)
- cache_stats_report
 - cache_stats.c, [66](#)
 - cache_stats.h, [79](#)
- cache_stats_show
 - cache_stats.c, [67](#)
 - cache_stats.h, [79](#)
- CacheStats, [8](#)
 - allocated_memory, [8](#)
 - cache_stats.h, [71](#)
 - del_counter, [8](#)
 - evict_counter, [8](#)
 - get_counter, [8](#)
 - key_counter, [9](#)
 - put_counter, [9](#)
- clean_socket
 - cache_server_utils.c, [165](#)
 - cache_server_utils.h, [179](#)
- cleaning
 - ClientData, [10](#)
- ClientData, [9](#)
 - cleaning, [10](#)
 - command, [10](#)
 - key, [10](#)
 - key_size, [10](#)
 - key_size_buffer, [10](#)
 - parsing_index, [10](#)
 - parsing_stage, [10](#)
 - socket, [10](#)
 - value, [11](#)
 - value_size, [11](#)
 - value_size_buffer, [11](#)
- Command
 - cache_server_models.h, [162](#)
- command
 - ClientData, [10](#)
- construct_new_client_epoll
 - cache_server_utils.c, [165](#)
 - cache_server_utils.h, [179](#)
- Counter
 - atom_counter.h, [120](#)
- counter
 - AtomCounter, [6](#)
- COUNTER_FORMAT
 - atom_counter.h, [120](#)
- create_error_lookup_result
 - results.c, [125](#)
 - results.h, [129](#)
- create_miss_lookup_result
 - results.c, [125](#)
 - results.h, [129](#)
- create_new_client_data
 - cache_server_utils.c, [166](#)
 - cache_server_utils.h, [180](#)
- create_ok_lookup_result
 - results.c, [125](#)

- results.h, [129](#)
- create_server_socket
 - server_starter_utils.c, [190](#)
 - server_starter_utils.h, [195](#)
- DEBUG
 - cache.h, [50](#)
- DEFAULT_PORT
 - server_starter_utils.c, [190](#)
 - server_starter_utils.h, [195](#)
- dek_hash
 - hash.c, [87](#)
 - hash.h, [89](#)
- DEL
 - cache_server_models.h, [162](#)
- del
 - StatsReport, [16](#)
- del_counter
 - CacheStats, [8](#)
- delete_client_data
 - cache_server_utils.c, [166](#)
 - cache_server_utils.h, [180](#)
- drop_client
 - cache_server_utils.c, [167](#)
 - cache_server_utils.h, [180](#)
- dynalloc
 - dynalloc.c, [83](#)
 - dynalloc.h, [86](#)
- dynalloc.c
 - dynalloc, [83](#)
 - DYNALLOC_FAIL_RATE, [83](#)
 - MAX_ATTEMPTS, [83](#)
 - SIZE_GOAL_FACTOR, [83](#)
- dynalloc.h
 - Cache, [86](#)
 - dynalloc, [86](#)
- dynalloc/dynalloc.c, [83, 84](#)
- dynalloc/dynalloc.h, [85, 87](#)
- DYNALLOC_FAIL_RATE
 - dynalloc.c, [83](#)
- EBIG
 - cache_server_models.h, [163](#)
- EBINARY
 - cache_server_models.h, [162](#)
- EINVALID
 - cache_server_models.h, [162](#)
- ENOTFOUND
 - cache_server_models.h, [162](#)
- ERROR
 - results.h, [128](#)
- EUNK
 - cache_server_models.h, [163](#)
- evict
 - StatsReport, [17](#)
- evict_counter
 - CacheStats, [8](#)
- exec_server
 - server_starter_utils.c, [191](#)
- server_starter_utils.h, [196](#)
- GET
 - cache_server_models.h, [162](#)
- get
 - StatsReport, [17](#)
- get_counter
 - CacheStats, [8](#)
- GIGABYTE
 - main.c, [20](#)
 - main2.c, [26](#)
- GREEN
 - cache_server.c, [155](#)
- handle_request
 - cache_server_utils.c, [167](#)
 - cache_server_utils.h, [181](#)
- hash.c
 - dek_hash, [87](#)
 - kr_hash, [88](#)
- hash.h
 - dek_hash, [89](#)
 - kr_hash, [89](#)
- hash_function
 - Cache, [7](#)
- hash_node
 - LRUNode, [14](#)
- HashFunction
 - cache.h, [50](#)
- hashmap/hash.c, [87, 88](#)
- hashmap/hash.h, [88, 89](#)
- hashmap/hashnode.c, [90, 100](#)
- hashmap/hashnode.h, [102, 113](#)
- HashNode, [11](#)
 - hashnode.h, [103](#)
 - key, [12](#)
 - key_size, [12](#)
 - lru.h, [138](#)
 - lrunode.h, [150](#)
 - next, [12](#)
 - prev, [12](#)
 - prio, [12](#)
 - val, [12](#)
 - val_size, [12](#)
- hashnode.c
 - hashnode_clean, [91](#)
 - hashnode_create, [91](#)
 - hashnode_destroy, [92](#)
 - hashnode_get_key, [92](#)
 - hashnode_get_key_size, [93](#)
 - hashnode_get_next, [93](#)
 - hashnode_get_prev, [94](#)
 - hashnode_get_prio, [94](#)
 - hashnode_get_val, [94](#)
 - hashnode_get_val_size, [95](#)
 - hashnode_keys_equal, [95](#)
 - hashnode_lookup, [95](#)
 - hashnode_lookup_node, [96](#)
 - hashnode_set_key, [97](#)

- hashnode_set_key_size, 97
- hashnode_set_next, 98
- hashnode_set_prev, 98
- hashnode_set_prio, 98
- hashnode_set_val, 99
- hashnode_set_val_size, 99
- hashnode.h
 - HashNode, 103
 - hashnode_clean, 104
 - hashnode_create, 104
 - hashnode_destroy, 105
 - hashnode_get_key, 106
 - hashnode_get_key_size, 106
 - hashnode_get_next, 106
 - hashnode_get_prev, 107
 - hashnode_get_prio, 107
 - hashnode_get_val, 108
 - hashnode_get_val_size, 108
 - hashnode_lookup, 108
 - hashnode_lookup_node, 109
 - hashnode_set_key, 109
 - hashnode_set_key_size, 110
 - hashnode_set_next, 110
 - hashnode_set_prev, 111
 - hashnode_set_prio, 111
 - hashnode_set_val, 112
 - hashnode_set_val_size, 112
 - LRUNode, 103
- hashnode_clean
 - hashnode.c, 91
 - hashnode.h, 104
- hashnode_create
 - hashnode.c, 91
 - hashnode.h, 104
- hashnode_destroy
 - hashnode.c, 92
 - hashnode.h, 105
- hashnode_get_key
 - hashnode.c, 92
 - hashnode.h, 106
- hashnode_get_key_size
 - hashnode.c, 93
 - hashnode.h, 106
- hashnode_get_next
 - hashnode.c, 93
 - hashnode.h, 106
- hashnode_get_prev
 - hashnode.c, 94
 - hashnode.h, 107
- hashnode_get_prio
 - hashnode.c, 94
 - hashnode.h, 107
- hashnode_get_val
 - hashnode.c, 94
 - hashnode.h, 108
- hashnode_get_val_size
 - hashnode.c, 95
 - hashnode.h, 108
- hashnode_keys_equal
 - hashnode.c, 95
- hashnode_lookup
 - hashnode.c, 95
 - hashnode.h, 108
- hashnode_lookup_node
 - hashnode.c, 96
 - hashnode.h, 109
- hashnode_set_key
 - hashnode.c, 97
 - hashnode.h, 109
- hashnode_set_key_size
 - hashnode.c, 97
 - hashnode.h, 110
- hashnode_set_next
 - hashnode.c, 98
 - hashnode.h, 110
- hashnode_set_prev
 - hashnode.c, 98
 - hashnode.h, 111
- hashnode_set_prio
 - hashnode.c, 98
 - hashnode.h, 111
- hashnode_set_val
 - hashnode.c, 99
 - hashnode.h, 112
- hashnode_set_val_size
 - hashnode.c, 99
 - hashnode.h, 112
- helpers/atom_counter.c, 115, 118
- helpers/atom_counter.h, 119, 123
- helpers/quit.c, 123, 124
- helpers/quit.h, 124
- helpers/results.c, 125, 127
- helpers/results.h, 127, 130
- key
 - ClientData, 10
 - HashNode, 12
 - StatsReport, 17
- KEY_COUNT
 - main.c, 20
 - main2.c, 26
 - main3.c, 32
- key_counter
 - CacheStats, 9
- KEY_SIZE
 - main.c, 20
 - main2.c, 26
 - main3.c, 32
- key_size
 - ClientData, 10
 - HashNode, 12
- key_size_buffer
 - ClientData, 10
- kr_hash
 - cache.h, 57
 - hash.c, 88
 - hash.h, 89

- least_recent
 - LRUQueue, 15
- LENGTH_PREFIX_SIZE
 - cache_server_models.h, 162
- lock
 - AtomCounter, 6
 - LRUQueue, 15
- lookup_result_get_size
 - results.c, 126
 - results.h, 129
- lookup_result_get_value
 - results.c, 126
 - results.h, 129
- lookup_result_is_error
 - results.c, 126
 - results.h, 129
- lookup_result_is_miss
 - results.c, 126
 - results.h, 130
- lookup_result_is_ok
 - results.c, 126
 - results.h, 130
- LookupResult, 13
 - ptr, 13
 - results.h, 128
 - size, 13
 - status, 13
- lru.c
 - lru_queue_create, 132
 - lru_queue_delete_node, 132
 - lru_queue_destroy, 133
 - lru_queue_get_least_recent, 133
 - lru_queue_lock, 134
 - lru_queue_node_clean, 134
 - lru_queue_set_most_recent, 134
 - lru_queue_unlock, 135
- lru.h
 - HashNode, 138
 - lru_queue_create, 138
 - lru_queue_delete_node, 139
 - lru_queue_destroy, 139
 - lru_queue_get_least_recent, 140
 - lru_queue_lock, 140
 - lru_queue_node_clean, 141
 - lru_queue_set_most_recent, 141
 - lru_queue_unlock, 142
 - LRUNode, 138
 - LRUQueue, 138
- lru/lru.c, 131, 136
- lru/lru.h, 138, 142
- lru/lrunode.c, 143, 148
- lru/lrunode.h, 149, 154
- lru_node_is_clean
 - lrunode.c, 144
 - lrunode.h, 150
- lru_queue_create
 - lru.c, 132
 - lru.h, 138
- lru_queue_delete_node
 - lru.c, 132
 - lru.h, 139
- lru_queue_destroy
 - lru.c, 133
 - lru.h, 139
- lru_queue_get_least_recent
 - lru.c, 133
 - lru.h, 140
- lru_queue_lock
 - lru.c, 134
 - lru.h, 140
- lru_queue_node_clean
 - lru.c, 134
 - lru.h, 141
- lru_queue_set_most_recent
 - lru.c, 134
 - lru.h, 141
- lru_queue_unlock
 - lru.c, 135
 - lru.h, 142
- LRUNode, 13
 - bucket_number, 14
 - hash_node, 14
 - hashnode.h, 103
 - lru.h, 138
 - lrunode.h, 150
 - next, 14
 - prev, 14
- lrunode.c
 - lru_node_is_clean, 144
 - lrunode_create, 145
 - lrunode_destroy, 145
 - lrunode_get_bucket_number, 145
 - lrunode_get_hash_node, 146
 - lrunode_get_next, 146
 - lrunode_get_prev, 146
 - lrunode_set_bucket_number, 147
 - lrunode_set_hash_node, 147
 - lrunode_set_next, 147
 - lrunode_set_prev, 148
 - PRINT, 144
- lrunode.h
 - Cache, 150
 - HashNode, 150
 - lru_node_is_clean, 150
 - LRUNode, 150
 - lrunode_create, 150
 - lrunode_destroy, 151
 - lrunode_get_bucket_number, 151
 - lrunode_get_hash_node, 151
 - lrunode_get_next, 152
 - lrunode_get_prev, 152
 - lrunode_set_bucket_number, 152
 - lrunode_set_hash_node, 153
 - lrunode_set_next, 153
 - lrunode_set_prev, 153
- lrunode_create

- lrunode.c, [145](#)
- lrunode.h, [150](#)
- lrunode_destroy
 - lrunode.c, [145](#)
 - lrunode.h, [151](#)
- lrunode_get_bucket_number
 - lrunode.c, [145](#)
 - lrunode.h, [151](#)
- lrunode_get_hash_node
 - lrunode.c, [146](#)
 - lrunode.h, [151](#)
- lrunode_get_next
 - lrunode.c, [146](#)
 - lrunode.h, [152](#)
- lrunode_get_prev
 - lrunode.c, [146](#)
 - lrunode.h, [152](#)
- lrunode_set_bucket_number
 - lrunode.c, [147](#)
 - lrunode.h, [152](#)
- lrunode_set_hash_node
 - lrunode.c, [147](#)
 - lrunode.h, [153](#)
- lrunode_set_next
 - lrunode.c, [147](#)
 - lrunode.h, [153](#)
- lrunode_set_prev
 - lrunode.c, [148](#)
 - lrunode.h, [153](#)
- LRUQueue, [14](#)
 - least_recent, [15](#)
 - lock, [15](#)
 - lru.h, [138](#)
 - most_recent, [15](#)
- main
 - cache_server.c, [156](#)
 - main.c, [21](#)
 - main2.c, [27](#)
 - main3.c, [33](#)
 - server_starter.c, [189](#)
- main.c
 - GIGABYTE, [20](#)
 - KEY_COUNT, [20](#)
 - KEY_SIZE, [20](#)
 - main, [21](#)
 - MEGABYTE, [20](#)
 - MEMORY_LIMIT, [20](#)
 - set_memory_limit, [21](#)
 - thread_func, [21](#)
 - turnstile, [23](#)
 - VAL_COUNT, [20](#)
 - VAL_SIZE, [20](#)
- main2
 - main3.c, [33](#)
- main2.c
 - GIGABYTE, [26](#)
 - KEY_COUNT, [26](#)
 - KEY_SIZE, [26](#)
- main, [27](#)
- MEGABYTE, [26](#)
- MEMORY_LIMIT, [26](#)
- set_memory_limit, [27](#)
- thread_func, [28](#)
- turnstile, [29](#)
- VAL_COUNT, [27](#)
- VAL_SIZE, [27](#)
- main3.c
 - KEY_COUNT, [32](#)
 - KEY_SIZE, [32](#)
 - main, [33](#)
 - main2, [33](#)
 - VAL_COUNT, [32](#)
 - VAL_SIZE, [32](#)
- MAX_ATTEMPTS
 - dynalloc.c, [83](#)
- MEGABYTE
 - main.c, [20](#)
 - main2.c, [26](#)
- MEMORY_LIMIT
 - main.c, [20](#)
 - main2.c, [26](#)
- memory_limit
 - Args, [5](#)
- MISS
 - results.h, [128](#)
- most_recent
 - LRUQueue, [15](#)
- next
 - HashNode, [12](#)
 - LRUNode, [14](#)
- num_buckets
 - Cache, [7](#)
- num_threads
 - Args, [5](#)
 - ServerArgs, [15](#)
- num_zones
 - Cache, [7](#)
- OK
 - results.h, [128](#)
- OKAY
 - cache_server_models.h, [162](#)
- ORANGE
 - cache_server.c, [155](#)
- parse_arguments
 - server_starter_utils.c, [191](#)
 - server_starter_utils.h, [196](#)
- parse_request
 - cache_server_utils.c, [169](#)
 - cache_server_utils.h, [182](#)
- PARSING_COMMAND
 - cache_server_models.h, [162](#)
- PARSING_FINISHED
 - cache_server_models.h, [162](#)
- parsing_index

- ClientData, 10
- PARSING_KEY
 - cache_server_models.h, 162
- PARSING_KEY_LEN
 - cache_server_models.h, 162
- parsing_stage
 - ClientData, 10
- PARSING_VALUE
 - cache_server_models.h, 162
- PARSING_VALUE_LEN
 - cache_server_models.h, 162
- ParsingStage
 - cache_server_models.h, 162
- port
 - Args, 5
- prev
 - HashNode, 12
 - LRUNode, 14
- PRINT
 - cache.h, 50
 - lrunode.c, 144
- prio
 - HashNode, 12
- ptr
 - LookupResult, 13
- PUT
 - cache_server_models.h, 162
- put
 - StatsReport, 17
- put_counter
 - CacheStats, 9
- queue
 - Cache, 7
- quit
 - quit.c, 123
 - quit.h, 124
- quit.c
 - quit, 123
- quit.h
 - quit, 124
- reconstruct_client_epoll
 - cache_server_utils.c, 171
 - cache_server_utils.h, 184
- recv_client
 - cache_server_utils.c, 171
 - cache_server_utils.h, 185
- RED
 - cache_server.c, 155
- RESET
 - cache_server.c, 155
- reset_client_data
 - cache_server_utils.c, 172
 - cache_server_utils.h, 185
- Response
 - cache_server_models.h, 162
- results.c
 - create_error_lookup_result, 125
 - create_miss_lookup_result, 125
 - create_ok_lookup_result, 125
 - lookup_result_get_size, 126
 - lookup_result_get_value, 126
 - lookup_result_is_error, 126
 - lookup_result_is_miss, 126
 - lookup_result_is_ok, 126
- results.h
 - create_error_lookup_result, 129
 - create_miss_lookup_result, 129
 - create_ok_lookup_result, 129
 - ERROR, 128
 - lookup_result_get_size, 129
 - lookup_result_get_value, 129
 - lookup_result_is_error, 129
 - lookup_result_is_miss, 130
 - lookup_result_is_ok, 130
 - LookupResult, 128
 - MISS, 128
 - OK, 128
 - Status, 128
- send_client
 - cache_server_utils.c, 172
 - cache_server_utils.h, 186
- server/cache_server.c, 155, 158
- server/cache_server_models.h, 161, 163
- server/cache_server_utils.c, 164, 173
- server/cache_server_utils.h, 178, 186
- server/server_starter.c, 188, 189
- server/server_starter_utils.c, 190, 193
- server/server_starter_utils.h, 195, 198
- server_epoll
 - ServerArgs, 16
 - ThreadArgs, 18
- server_socket
 - ServerArgs, 16
 - ThreadArgs, 18
- server_starter.c
 - main, 189
- server_starter_utils.c
 - BACKLOG_SIZE, 190
 - create_server_socket, 190
 - DEFAULT_PORT, 190
 - exec_server, 191
 - parse_arguments, 191
 - set_memory_limit, 192
- server_starter_utils.h
 - create_server_socket, 195
 - DEFAULT_PORT, 195
 - exec_server, 196
 - parse_arguments, 196
 - set_memory_limit, 197
- ServerArgs, 15
 - cache, 15
 - num_threads, 15
 - server_epoll, 16
 - server_socket, 16
- set_memory_limit

- main.c, [21](#)
- main2.c, [27](#)
- server_starter_utils.c, [192](#)
- server_starter_utils.h, [197](#)
- size
 - LookupResult, [13](#)
- SIZE_GOAL_FACTOR
 - dynalloc.c, [83](#)
- socket
 - ClientData, [10](#)
- SOFT_RED
 - cache_server.c, [156](#)
- start_server
 - cache_server.c, [156](#)
- STATS
 - cache_server_models.h, [162](#)
- stats
 - Cache, [7](#)
- STATS_COUNT
 - cache_stats.h, [71](#)
- STATS_MESSAGE_LENGTH
 - cache_stats.h, [71](#)
- stats_report_stringify
 - cache_stats.c, [67](#)
 - cache_stats.h, [80](#)
- StatsReport, [16](#)
 - allocated_memory, [16](#)
 - cache_stats.h, [71](#)
 - del, [16](#)
 - evict, [17](#)
 - get, [17](#)
 - key, [17](#)
 - put, [17](#)
- Status
 - results.h, [128](#)
- status
 - LookupResult, [13](#)
- thread_func
 - main.c, [21](#)
 - main2.c, [28](#)
- thread_id
 - ThreadArgs, [18](#)
- thread_number
 - ThreadArgs, [18](#)
- ThreadArgs, [17](#)
 - cache, [18](#)
 - server_epoll, [18](#)
 - server_socket, [18](#)
 - thread_id, [18](#)
 - thread_number, [18](#)
- TRASH_BUFFER_SIZE
 - cache_server_utils.c, [165](#)
- turnstile
 - main.c, [23](#)
 - main2.c, [29](#)
- val
 - HashNode, [12](#)
- VAL_COUNT
 - main.c, [20](#)
 - main2.c, [27](#)
 - main3.c, [32](#)
- VAL_SIZE
 - main.c, [20](#)
 - main2.c, [27](#)
 - main3.c, [32](#)
- val_size
 - HashNode, [12](#)
- value
 - ClientData, [11](#)
- value_size
 - ClientData, [11](#)
- value_size_buffer
 - ClientData, [11](#)
- working_thread
 - cache_server.c, [157](#)
- zone_locks
 - Cache, [7](#)
- ZONES_FACTOR
 - cache.c, [36](#)