

UMS: Library

Generated by Doxygen 1.9.2



<b>1 Data Structure Index</b>	<b>1</b>
1.1 Data Structures	1
<b>2 File Index</b>	<b>3</b>
2.1 File List	3
<b>3 Data Structure Documentation</b>	<b>5</b>
3.1 hlist_head Struct Reference	5
3.2 hlist_node Struct Reference	5
3.3 list_head Struct Reference	5
3.3.1 Detailed Description	6
3.4 list_params Struct Reference	6
3.4.1 Detailed Description	6
3.4.2 Field Documentation	6
3.4.2.1 size	6
3.4.2.2 state	6
3.4.2.3 worker_count	7
3.4.2.4 workers	7
3.5 scheduler_params Struct Reference	7
3.5.1 Detailed Description	7
3.5.2 Field Documentation	7
3.5.2.1 clid	7
3.5.2.2 core_id	8
3.5.2.3 entry_point	8
3.5.2.4 sid	8
3.6 ums_completion_list Struct Reference	8
3.6.1 Detailed Description	8
3.6.2 Field Documentation	8
3.6.2.1 count	9
3.7 ums_completion_list_node Struct Reference	9
3.7.1 Detailed Description	9
3.7.2 Field Documentation	9
3.7.2.1 clid	9
3.7.2.2 list_params	9
3.7.2.3 state	10
3.7.2.4 worker_count	10
3.8 ums_scheduler Struct Reference	10
3.8.1 Detailed Description	10
3.8.2 Field Documentation	10
3.8.2.1 list_params	10
3.8.2.2 sched_params	11
3.8.2.3 tid	11
3.8.2.4 wid	11

3.9 ums_scheduler_list Struct Reference	11
3.9.1 Detailed Description	11
3.9.2 Field Documentation	11
3.9.2.1 count	12
3.10 ums_worker Struct Reference	12
3.10.1 Detailed Description	12
3.10.2 Field Documentation	12
3.10.2.1 state	12
3.10.2.2 wid	12
3.10.2.3 worker_params	13
3.11 ums_worker_list Struct Reference	13
3.11.1 Detailed Description	13
3.11.2 Field Documentation	13
3.11.2.1 count	13
3.12 worker_params Struct Reference	13
3.12.1 Detailed Description	14
3.12.2 Field Documentation	14
3.12.2.1 clid	14
3.12.2.2 entry_point	14
3.12.2.3 function_args	14
3.12.2.4 stack_addr	14
3.12.2.5 stack_size	14
<b>4 File Documentation</b>	<b>15</b>
4.1 const.h File Reference	15
4.1.1 Detailed Description	17
4.1.2 Enumeration Type Documentation	17
4.1.2.1 state	17
4.1.2.2 worker_status	18
4.2 const.h	18
4.3 list.h File Reference	19
4.3.1 Detailed Description	20
4.3.2 Macro Definition Documentation	21
4.3.2.1 __list_for_each	21
4.3.2.2 container_of	21
4.3.2.3 hlist_for_each	21
4.3.2.4 hlist_for_each_entry	22
4.3.2.5 hlist_for_each_entry_continue	22
4.3.2.6 hlist_for_each_entry_from	22
4.3.2.7 hlist_for_each_entry_safe	23
4.3.2.8 hlist_for_each_safe	23
4.3.2.9 INIT_LIST_HEAD	23

4.3.2.10 list_entry . . . . .	23
4.3.2.11 list_for_each . . . . .	24
4.3.2.12 list_for_each_entry . . . . .	24
4.3.2.13 list_for_each_entry_continue . . . . .	24
4.3.2.14 list_for_each_entry_reverse . . . . .	24
4.3.2.15 list_for_each_entry_safe . . . . .	25
4.3.2.16 list_for_each_entry_safe_continue . . . . .	25
4.3.2.17 list_for_each_entry_safe_reverse . . . . .	25
4.3.2.18 list_for_each_prev . . . . .	26
4.3.2.19 list_for_each_safe . . . . .	26
4.3.2.20 list_prepare_entry . . . . .	26
4.3.2.21 offsetof . . . . .	26
4.4 list.h . . . . .	27
4.5 ums_lib.c File Reference . . . . .	30
4.5.1 Detailed Description . . . . .	32
4.5.2 Function Documentation . . . . .	33
4.5.2.1 check_if_completion_list_exists() . . . . .	33
4.5.2.2 check_if_scheduler_exists() . . . . .	34
4.5.2.3 check_if_worker_exists() . . . . .	34
4.5.2.4 cleanup() . . . . .	34
4.5.2.5 close_device() . . . . .	35
4.5.2.6 open_device() . . . . .	35
4.5.2.7 ums_create_completion_list() . . . . .	35
4.5.2.8 ums_create_scheduler() . . . . .	35
4.5.2.9 ums_create_worker_thread() . . . . .	36
4.5.2.10 ums_dequeue_completion_list_items() . . . . .	36
4.5.2.11 ums_enter() . . . . .	37
4.5.2.12 ums_enter_scheduling_mode() . . . . .	37
4.5.2.13 ums_execute_thread() . . . . .	37
4.5.2.14 ums_exit() . . . . .	38
4.5.2.15 ums_exit_scheduling_mode() . . . . .	38
4.5.2.16 ums_get_next_worker_thread() . . . . .	38
4.5.2.17 ums_thread_exit() . . . . .	38
4.5.2.18 ums_thread_pause() . . . . .	39
4.5.2.19 ums_thread_yield() . . . . .	39
4.5.3 Variable Documentation . . . . .	39
4.5.3.1 completion_lists . . . . .	39
4.5.3.2 schedulers . . . . .	40
4.5.3.3 workers . . . . .	40
4.6 ums_lib.h File Reference . . . . .	40
4.6.1 Detailed Description . . . . .	42
4.6.2 Function Documentation . . . . .	43

---

4.6.2.1	<code>check_if_completion_list_exists()</code>	43
4.6.2.2	<code>check_if_scheduler_exists()</code>	43
4.6.2.3	<code>check_if_worker_exists()</code>	43
4.6.2.4	<code>cleanup()</code>	44
4.6.2.5	<code>close_device()</code>	44
4.6.2.6	<code>open_device()</code>	44
4.6.2.7	<code>ums_create_completion_list()</code>	44
4.6.2.8	<code>ums_create_worker_thread()</code>	44
4.6.2.9	<code>ums_dequeue_completion_list_items()</code>	45
4.6.2.10	<code>ums_enter()</code>	45
4.6.2.11	<code>ums_enter_scheduling_mode()</code>	45
4.6.2.12	<code>ums_execute_thread()</code>	46
4.6.2.13	<code>ums_exit()</code>	46
4.6.2.14	<code>ums_exit_scheduling_mode()</code>	46
4.6.2.15	<code>ums_get_next_worker_thread()</code>	47
4.6.2.16	<code>ums_thread_exit()</code>	47
4.6.2.17	<code>ums_thread_pause()</code>	47
4.7	<code>ums_lib.h</code>	48
	<b>Index</b>	<b>49</b>

# Chapter 1

## Data Structure Index

### 1.1 Data Structures

Here are the data structures with brief descriptions:

<a href="#">hlist_head</a>	5
<a href="#">hlist_node</a>	5
<a href="#">list_head</a>	5
<a href="#">list_params</a>	
Parameters that are created by the scheduler and passed to dequeue the completion list items	6
<a href="#">scheduler_params</a>	
Parameters that are passed in order to create a scheduler	7
<a href="#">ums_completion_list</a>	
The list of the completion lists created by the process	8
<a href="#">ums_completion_list_node</a>	
Represents a node in the <a href="#">ums_completion_list</a>	9
<a href="#">ums_scheduler</a>	
Represents a node in the <a href="#">ums_scheduler_list</a>	10
<a href="#">ums_scheduler_list</a>	
The list of the schedulers created by the process	11
<a href="#">ums_worker</a>	
Represents a node in the <a href="#">ums_worker_list</a>	12
<a href="#">ums_worker_list</a>	
The list of the worker threads created by the process	13
<a href="#">worker_params</a>	
Parameters that are passed in order to create a worker thread	13





## Chapter 2

# File Index

### 2.1 File List

Here is a list of all documented files with brief descriptions:

<a href="#">const.h</a>	Set of data structures and other constant variables used by UMS library . . . . .	15
<a href="#">list.h</a>	Implementation of the Linux kernel linked list and hash list data structures for user space . . .	19
<a href="#">ums_lib.c</a>	Contains implementations of the essential UMS library functions . . . . .	30
<a href="#">ums_lib.h</a>	The header that contains essential UMS library functions and has to be included by the user in order to use the UMS library . . . . .	40



## Chapter 3

# Data Structure Documentation

### 3.1 hlist\_head Struct Reference

#### Data Fields

- struct [hlist\\_node](#) \* **first**

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

- [list.h](#)

### 3.2 hlist\_node Struct Reference

#### Data Fields

- struct [hlist\\_node](#) \* **next**
- struct [hlist\\_node](#) \*\* **pprev**

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

- [list.h](#)

### 3.3 list\_head Struct Reference

```
#include <list.h>
```

#### Data Fields

- struct [list\\_head](#) \* **next**
- struct [list\\_head](#) \* **prev**

### 3.3.1 Detailed Description

Simple doubly linked list implementation.

Some of the internal functions ("\_\_xxx") are useful when manipulating whole lists rather than single entries, as sometimes we already know the next/prev entries and we can generate better code by using them directly rather than using the generic single-entry routines.

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

- [list.h](#)

## 3.4 list\_params Struct Reference

Parameters that are created by the scheduler and passed to dequeue the completion list items

```
#include <const.h>
```

### Data Fields

- unsigned int [size](#)
- unsigned int [worker\\_count](#)
- [state\\_t](#) [state](#)
- [ums\\_wid\\_t](#) [workers](#) []

### 3.4.1 Detailed Description

Parameters that are created by the scheduler and passed to dequeue the completion list items

### 3.4.2 Field Documentation

#### 3.4.2.1 size

```
unsigned int list_params::size
```

Size of the worker thread array

#### 3.4.2.2 state

```
state_t list_params::state
```

Tracks the state of the completion list which is set by the kernel module after a dequeue call

### 3.4.2.3 worker\_count

```
unsigned int list_params::worker_count
```

Tracks the quantity of the available workers and used as state indicator for scheduler to perform a new dequeue call

### 3.4.2.4 workers

```
ums_wid_t list_params::workers[]
```

Array of worker threads. Stores ID of worker threads in case they are available to be scheduled (when worker thread is finished, scheduler replaces ID with -1 value)

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

- [const.h](#)

## 3.5 scheduler\_params Struct Reference

Parameters that are passed in order to create a scheduler

```
#include <const.h>
```

### Data Fields

- unsigned long [entry\\_point](#)
- [ums\\_clid\\_t](#) clid
- [ums\\_sid\\_t](#) sid
- int [core\\_id](#)

### 3.5.1 Detailed Description

Parameters that are passed in order to create a scheduler

### 3.5.2 Field Documentation

#### 3.5.2.1 clid

```
ums_clid_t scheduler_params::clid
```

ID of the completion list that is assigned to the scheduler

### 3.5.2.2 core\_id

```
int scheduler_params::core_id
```

ID of the CPU core that is assigned to the scheduler (It is handled automatically by the library, no user input required)

### 3.5.2.3 entry\_point

```
unsigned long scheduler_params::entry_point
```

Function pointer and an entry point set by a user, that serves as a starting point of the scheduler. It is a scheduling function that determines the next thread to be scheduled

### 3.5.2.4 sid

```
ums_sid_t scheduler_params::sid
```

ID of the scheduler which is set by the kernel module

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

- [const.h](#)

## 3.6 ums\_completion\_list Struct Reference

The list of the completion lists created by the process

```
#include <ums_lib.h>
```

### Data Fields

- struct [list\\_head](#) list
- unsigned int [count](#)

### 3.6.1 Detailed Description

The list of the completion lists created by the process

### 3.6.2 Field Documentation

### 3.6.2.1 count

```
unsigned int ums_completion_list::count
```

Number of completion lists created

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

- [ums\\_lib.h](#)

## 3.7 ums\_completion\_list\_node Struct Reference

Represents a node in the [ums\\_completion\\_list](#)

```
#include <ums_lib.h>
```

### Data Fields

- [ums\\_clid\\_t](#) clid
- [state\\_t](#) state
- unsigned int [worker\\_count](#)
- struct [list\\_head](#) list
- [list\\_params\\_t](#)\* list\_params

### 3.7.1 Detailed Description

Represents a node in the [ums\\_completion\\_list](#)

### 3.7.2 Field Documentation

#### 3.7.2.1 clid

```
ums_clid_t ums_completion_list_node::clid
```

Completion list ID

#### 3.7.2.2 list\_params

```
list_params_t* ums_completion_list_node::list_params
```

Parameters that are created by the scheduler and passed to dequeue the completion list items [list\\_params](#)

### 3.7.2.3 state

```
state_t ums_completion_list_node::state
```

State of the completion list

### 3.7.2.4 worker\_count

```
unsigned int ums_completion_list_node::worker_count
```

Number of worker threads assigned to the completion list

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

- [ums\\_lib.h](#)

## 3.8 ums\_scheduler Struct Reference

Represents a node in the [ums\\_scheduler\\_list](#)

```
#include <ums_lib.h>
```

### Data Fields

- struct [list\\_head](#) list
- pthread\_t tid
- [ums\\_wid\\_t](#) wid
- [scheduler\\_params\\_t](#) \* sched\_params
- [list\\_params\\_t](#) \* list\_params

### 3.8.1 Detailed Description

Represents a node in the [ums\\_scheduler\\_list](#)

### 3.8.2 Field Documentation

#### 3.8.2.1 list\_params

```
list_params_t* ums_scheduler::list_params
```

Parameters that are created by the scheduler and passed to dequeue the completion list items [list\\_params](#)



### 3.8.2.2 sched\_params

```
scheduler_params_t* ums_scheduler::sched_params
```

Parameters that are passed in order to create a scheduler [scheduler\\_params](#)

### 3.8.2.3 tid

```
pthread_t ums_scheduler::tid
```

Pthread ID

### 3.8.2.4 wid

```
ums_wid_t ums_scheduler::wid
```

Worker thread ID

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

- [ums\\_lib.h](#)

## 3.9 ums\_scheduler\_list Struct Reference

The list of the schedulers created by the process

```
#include <ums_lib.h>
```

### Data Fields

- struct [list\\_head](#) **list**
- unsigned int [count](#)

### 3.9.1 Detailed Description

The list of the schedulers created by the process

### 3.9.2 Field Documentation

### 3.9.2.1 count

```
unsigned int ums_scheduler_list::count
```

Number of scheduler created

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

- [ums\\_lib.h](#)

## 3.10 ums\_worker Struct Reference

Represents a node in the [ums\\_worker\\_list](#)

```
#include <ums_lib.h>
```

### Data Fields

- [ums\\_wid\\_t](#) wid
- [state\\_t](#) state
- struct [list\\_head](#) list
- [worker\\_params\\_t](#) \* worker\_params

### 3.10.1 Detailed Description

Represents a node in the [ums\\_worker\\_list](#)

### 3.10.2 Field Documentation

#### 3.10.2.1 state

```
state_t ums_worker::state
```

State of worker thread's progress

#### 3.10.2.2 wid

```
ums_wid_t ums_worker::wid
```

Worker thread ID

### 3.10.2.3 worker\_params

```
worker_params_t* ums_worker::worker_params
```

Parameters that are passed in order to create a worker thread [worker\\_params](#)

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

- [ums\\_lib.h](#)

## 3.11 ums\_worker\_list Struct Reference

The list of the worker threads created by the process

```
#include <ums_lib.h>
```

### Data Fields

- struct [list\\_head](#) [list](#)
- unsigned int [count](#)

### 3.11.1 Detailed Description

The list of the worker threads created by the process

### 3.11.2 Field Documentation

#### 3.11.2.1 count

```
unsigned int ums_worker_list::count
```

Number of worker threads created

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

- [ums\\_lib.h](#)

## 3.12 worker\_params Struct Reference

Parameters that are passed in order to create a worker thread

```
#include <const.h>
```

## Data Fields

- unsigned long [entry\\_point](#)
- unsigned long [function\\_args](#)
- unsigned long [stack\\_size](#)
- unsigned long [stack\\_addr](#)
- [ums\\_clid\\_t](#) `clid`

### 3.12.1 Detailed Description

Parameters that are passed in order to create a worker thread

### 3.12.2 Field Documentation

#### 3.12.2.1 `clid`

```
ums_clid_t worker_params::clid
```

ID of the completion list where worker thread is assigned to

#### 3.12.2.2 `entry_point`

```
unsigned long worker_params::entry_point
```

Function pointer and an entry point set by a user, that serves as a starting point of the worker thread

#### 3.12.2.3 `function_args`

```
unsigned long worker_params::function_args
```

Pointer of the function arguments that are passed to the entry point/function

#### 3.12.2.4 `stack_addr`

```
unsigned long worker_params::stack_addr
```

Address of the stack allocated by the UMS library

#### 3.12.2.5 `stack_size`

```
unsigned long worker_params::stack_size
```

Stack size of the worker thread set by a user

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

- [const.h](#)

## Chapter 4

# File Documentation

### 4.1 const.h File Reference

Set of data structures and other constant variables used by UMS library.

```
#include <linux/ioctl.h>
```

#### Data Structures

- struct [list\\_params](#)  
*Parameters that are created by the scheduler and passed to dequeue the completion list items*
- struct [worker\\_params](#)  
*Parameters that are passed in order to create a worker thread*
- struct [scheduler\\_params](#)  
*Parameters that are passed in order to create a scheduler*

#### Macros

- `#define UMS_NAME "ums"`
- `#define UMS_DEVICE "/dev/ums"`
- `#define UMS_IOC_MAGIC 'R'`
- `#define UMS_ENTER _IO(UMS_IOC_MAGIC, 1)`
- `#define UMS_EXIT _IO(UMS_IOC_MAGIC, 2)`
- `#define UMS_CREATE_LIST _IO(UMS_IOC_MAGIC, 3)`
- `#define UMS_CREATE_WORKER _IOW(UMS_IOC_MAGIC, 4, unsigned long)`
- `#define UMS_ENTER_SCHEDULING_MODE _IOWR(UMS_IOC_MAGIC, 5, unsigned long)`
- `#define UMS_EXIT_SCHEDULING_MODE _IO(UMS_IOC_MAGIC, 6)`
- `#define UMS_EXECUTE_THREAD _IOW(UMS_IOC_MAGIC, 7, unsigned long)`
- `#define UMS_THREAD_YIELD _IOW(UMS_IOC_MAGIC, 8, unsigned long)`
- `#define UMS_DEQUEUE_COMPLETION_LIST_ITEMS _IOWR(UMS_IOC_MAGIC, 9, unsigned long)`
- `#define UMS_SUCCESS 0`  
*Successful execution.*
- `#define UMS_ERROR 1`  
*Error.*

- **#define UMS\_ERROR\_PROCESS\_NOT\_FOUND 1000**  
*Process is not managed by UMS kernel module.*
- **#define UMS\_ERROR\_PROCESS\_ALREADY\_EXISTS 1001**  
*Process is already managed by UMS kernel module.*
- **#define UMS\_ERROR\_COMPLETION\_LIST\_NOT\_FOUND 1002**  
*Completion list cannot be found.*
- **#define UMS\_ERROR\_SCHEDULER\_NOT\_FOUND 1003**  
*Scheduler cannot be found.*
- **#define UMS\_ERROR\_WORKER\_NOT\_FOUND 1004**  
*Worker thread cannot be found.*
- **#define UMS\_ERROR\_STATE\_RUNNING 1005**  
*The object is still running, thus cannot be modified, updated, deleted.*
- **#define UMS\_ERROR\_CMD\_IS\_NOT\_ISSUED\_BY\_MAIN\_THREAD 1006**  
*The command is not issued by the main process thread, e.g. [ums\\_exit\(\)](#)*
- **#define UMS\_ERROR\_WORKER\_ALREADY\_RUNNING 1007**  
*The worker thread is already running.*
- **#define UMS\_ERROR\_WRONG\_INPUT 1008**  
*Wrong input.*
- **#define UMS\_ERROR\_CMD\_IS\_NOT\_ISSUED\_BY\_SCHEDULER 1009**  
*The command is not issued by the scheduler.*
- **#define UMS\_ERROR\_CMD\_IS\_NOT\_ISSUED\_BY\_WORKER 1010**  
*The command is not issued by the worker.*
- **#define UMS\_ERROR\_WORKER\_ALREADY\_FINISHED 1011**  
*The worker thread has already finished execution.*
- **#define UMS\_ERROR\_NO\_AVAILABLE\_WORKERS 1012**  
*No worker threads are available.*
- **#define UMS\_ERROR\_COMPLETION\_LIST\_ALREADY\_FINISHED 1013**  
*All worker threads in the completion list have finished execution.*
- **#define UMS\_ERROR\_FAILED\_TO\_CREATE\_PROC\_ENTRY 1014**  
*Failed to create proc entry.*
- **#define UMS\_ERROR\_FAILED\_TO\_PROC\_OPEN 1015**  
*Failed to open proc entry.*
- **#define UMS\_ERROR\_COMPLETION\_LIST\_IS\_USED\_AND\_CANNOT\_BE\_MODIFIED 1016**  
*The completion list is being used, thus cannot be modified.*
- **#define UMS\_MIN\_STACK\_SIZE 4096**  
*The minimum stack size of the worker thread*

## Typedefs

- typedef enum [state](#) **state\_t**  
*States of processes, completion lists and threads (schedulers, worker threads)*
- typedef enum [worker\\_status](#) **worker\_status\_t**  
*Status of the worker thread Used as a parameter that is passed for pausing or completing the worker thread.*
- typedef unsigned int **ums\_sid\_t**  
*Scheduler ID*
- typedef unsigned int **ums\_wid\_t**  
*Worker thread ID*
- typedef unsigned int **ums\_clid\_t**  
*Completion list ID*
- typedef struct [list\\_params](#) **list\_params\_t**

*Parameters that are created by the scheduler and passed to dequeue the completion list items*

- typedef struct [worker\\_params](#) **worker\_params\_t**

*Parameters that are passed in order to create a worker thread*

- typedef struct [scheduler\\_params](#) **scheduler\_params\_t**

*Parameters that are passed in order to create a scheduler*

## Enumerations

- enum [state](#) { [IDLE](#) , [RUNNING](#) , [FINISHED](#) }

*States of processes, completion lists and threads (schedulers, worker threads)*

- enum [worker\\_status](#) { [PAUSE](#) , [FINISH](#) }

*Status of the worker thread Used as a parameter that is passed for pausing or completing the worker thread.*

### 4.1.1 Detailed Description

Set of data structures and other constant variables used by UMS library.

Copyright (C) 2021 Bektur Umarbaev [hrafnulf13@gmail.com](mailto:hrafnulf13@gmail.com)

This file is part of the User Mode thread Scheduling (UMS) library.

UMS library is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

UMS library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with UMS library. If not, see <http://www.gnu.org/licenses/>.

#### Author

Bektur Umarbaev [hrafnulf13@gmail.com](mailto:hrafnulf13@gmail.com)

#### Date

### 4.1.2 Enumeration Type Documentation

#### 4.1.2.1 state

enum [state](#)

States of processes, completion lists and threads (schedulers, worker threads)

## Enumerator

IDLE	Represents the state when worker thread is waiting to be scheduled; When scheduler waits or searches for available worker threads to run; Completion list has available worker threads to be scheduled
RUNNING	Represents the state when worker thread is scheduled and ran by the scheduler; When scheduler handles worker thread; Completion list is currently used and can't be modified
FINISHED	Represents the state when worker thread has been completed; When scheduler has completed all scheduling work with a completion list; All completion list's worker threads has been completed

## 4.1.2.2 worker\_status

```
enum worker_status
```

Status of the worker thread Used as a parameter that is passed for pausing or completing the worker thread.

## Enumerator

PAUSE	Used for pausing a worker thread: <code>ums_thread_pause() == ums_thread_yield(PAUSE)</code>
FINISH	Used for completing a worker thread: <code>ums_thread_exit() == ums_thread_yield(FINISH)</code>

## 4.2 const.h

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

```

1
29 #pragma once
30
31 #include <linux/ioctl.h>
32
33 /*
34  * Definitions
35  */
36
37 #define UMS_NAME          "ums"
38 #define UMS_DEVICE        "/dev/ums"
39 #define UMS_IOC_MAGIC     'R'
40
41 /*
42  * IOCTL definitions
43  */
44 #define UMS_ENTER          _IO(UMS_IOC_MAGIC, 1)
45 #define UMS_EXIT           _IO(UMS_IOC_MAGIC, 2)
46 #define UMS_CREATE_LIST   _IO(UMS_IOC_MAGIC, 3)
47 #define UMS_CREATE_WORKER _IOW(UMS_IOC_MAGIC, 4, unsigned long)
48 #define UMS_ENTER_SCHEDULING_MODE _IOWR(UMS_IOC_MAGIC, 5, unsigned long)
49 #define UMS_EXIT_SCHEDULING_MODE _IO(UMS_IOC_MAGIC, 6)
50 #define UMS_EXECUTE_THREAD _IOW(UMS_IOC_MAGIC, 7, unsigned long)
51 #define UMS_THREAD_YIELD  _IOW(UMS_IOC_MAGIC, 8, unsigned long)
52 #define UMS_DEQUEUE_COMPLETION_LIST_ITEMS _IOWR(UMS_IOC_MAGIC, 9, unsigned long)
53
54 /*
55  * Errors and return values
56  */
57 #define UMS_SUCCESS        0
58
59 #define UMS_ERROR          1
60
61 #define UMS_ERROR_PROCESS_NOT_FOUND 1000
62
63 #define UMS_ERROR_PROCESS_ALREADY_EXISTS 1001

```



```

61 #define UMS_ERROR_COMPLETION_LIST_NOT_FOUND 1002
62 #define UMS_ERROR_SCHEDULER_NOT_FOUND 1003
63 #define UMS_ERROR_WORKER_NOT_FOUND 1004
64 #define UMS_ERROR_STATE_RUNNING 1005
65 #define UMS_ERROR_CMD_IS_NOT_ISSUED_BY_MAIN_THREAD 1006
66 #define UMS_ERROR_WORKER_ALREADY_RUNNING 1007
67 #define UMS_ERROR_WRONG_INPUT 1008
68 #define UMS_ERROR_CMD_IS_NOT_ISSUED_BY_SCHEDULER 1009
69 #define UMS_ERROR_CMD_IS_NOT_ISSUED_BY_WORKER 1010
70 #define UMS_ERROR_WORKER_ALREADY_FINISHED 1011
71 #define UMS_ERROR_NO_AVAILABLE_WORKERS 1012
72 #define UMS_ERROR_COMPLETION_LIST_ALREADY_FINISHED 1013
73 #define UMS_ERROR_FAILED_TO_CREATE_PROC_ENTRY 1014
74 #define UMS_ERROR_FAILED_TO_PROC_OPEN 1015
75 #define UMS_ERROR_COMPLETION_LIST_IS_USED_AND_CANNOT_BE_MODIFIED 1016
76
81 #define UMS_MIN_STACK_SIZE 4096
82
87 typedef enum state {
88     IDLE,
89     RUNNING,
90     FINISHED
91 } state_t;
92
97 typedef enum worker_status {
98     PAUSE,
99     FINISH
100 } worker_status_t;
101
106 typedef unsigned int ums_sid_t;
107
112 typedef unsigned int ums_wid_t;
113
118 typedef unsigned int ums_clid_t;
119
124 typedef struct list_params {
125     unsigned int size;
126     unsigned int worker_count;
127     state_t state;
128     ums_wid_t workers[];
129 } list_params_t;
130
135 typedef struct worker_params {
136     unsigned long entry_point;
137     unsigned long function_args;
138     unsigned long stack_size;
139     unsigned long stack_addr;
140     ums_clid_t clid;
141 } worker_params_t;
142
147 typedef struct scheduler_params {
148     unsigned long entry_point;
149     ums_clid_t clid;
150     ums_sid_t sid;
151     int core_id;
152 } scheduler_params_t;

```

## 4.3 list.h File Reference

Implementation of the Linux kernel linked list and hash list data structures for user space.

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

## Data Structures

- struct [list\\_head](#)
- struct [hlist\\_head](#)
- struct [hlist\\_node](#)

### from other kernel headers

- #define [offsetof](#)(TYPE, MEMBER) ((size\_t) &((TYPE \*)0)->MEMBER)
- #define [container\\_of](#)(ptr, type, member)
- #define [LIST\\_POISON1](#) ((void \*) 0x00100100)
- #define [LIST\\_POISON2](#) ((void \*) 0x00200200)
- #define [LIST\\_HEAD\\_INIT](#)(name) { &(name), &(name) }
- #define [LIST\\_HEAD](#)(name) struct [list\\_head](#) name = [LIST\\_HEAD\\_INIT](#)(name)
- #define [INIT\\_LIST\\_HEAD](#)(ptr)
- #define [list\\_entry](#)(ptr, type, member) [container\\_of](#)(ptr, type, member)
- #define [list\\_for\\_each](#)(pos, head)
- #define [\\_\\_list\\_for\\_each](#)(pos, head) for (pos = (head)->next; pos != (head); pos = pos->next)
- #define [list\\_for\\_each\\_prev](#)(pos, head)
- #define [list\\_for\\_each\\_safe](#)(pos, n, head)
- #define [list\\_for\\_each\\_entry](#)(pos, head, member)
- #define [list\\_for\\_each\\_entry\\_reverse](#)(pos, head, member)
- #define [list\\_prepare\\_entry](#)(pos, head, member) ((pos) ? : [list\\_entry](#)(head, typeof(\*pos), member))
- #define [list\\_for\\_each\\_entry\\_continue](#)(pos, head, member)
- #define [list\\_for\\_each\\_entry\\_safe](#)(pos, n, head, member)
- #define [list\\_for\\_each\\_entry\\_safe\\_continue](#)(pos, n, head, member)
- #define [list\\_for\\_each\\_entry\\_safe\\_reverse](#)(pos, n, head, member)
- #define [HLIST\\_HEAD\\_INIT](#) { .first = NULL }
- #define [HLIST\\_HEAD](#)(name) struct [hlist\\_head](#) name = { .first = NULL }
- #define [INIT\\_HLIST\\_HEAD](#)(ptr) ((ptr)->first = NULL)
- #define [INIT\\_HLIST\\_NODE](#)(ptr) ((ptr)->next = NULL, (ptr)->pprev = NULL)
- #define [hlist\\_entry](#)(ptr, type, member) [container\\_of](#)(ptr,type,member)
- #define [hlist\\_for\\_each](#)(pos, head)
- #define [hlist\\_for\\_each\\_safe](#)(pos, n, head)
- #define [hlist\\_for\\_each\\_entry](#)(tpos, pos, head, member)
- #define [hlist\\_for\\_each\\_entry\\_continue](#)(tpos, pos, member)
- #define [hlist\\_for\\_each\\_entry\\_from](#)(tpos, pos, member)
- #define [hlist\\_for\\_each\\_entry\\_safe](#)(tpos, pos, n, head, member)

### 4.3.1 Detailed Description

Implementation of the Linux kernel linked list and hash list data structures for user space.

Copyright (C) 2021 Bektur Umarbaev [hrafnulf13@gmail.com](mailto:hrafnulf13@gmail.com)

This file is part of the User Mode thread Scheduling (UMS) library.

UMS library is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

UMS library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with UMS library. If not, see <http://www.gnu.org/licenses/>.

The following file is shared under GPL license and was downloaded from <http://www.mcs.anl.gov/~kazutomo/list/list.h> The file without modifications was used for the implementation of UMS library. Here by copyright, credits are attributed to Kazutomo Yoshii [kazutomo@mcs.anl.gov](mailto:kazutomo@mcs.anl.gov).

## Author

Bektur Umarbaev [hrafnulf13@gmail.com](mailto:hrafnulf13@gmail.com)

## Date

## 4.3.2 Macro Definition Documentation

### 4.3.2.1 \_\_list\_for\_each

```
#define __list_for_each(  
    pos,  
    head )    for (pos = (head)->next; pos != (head); pos = pos->next)
```

`__list_for_each` - iterate over a list @pos: the &struct [list\\_head](#) to use as a loop counter. @head: the head for your list.

This variant differs from [list\\_for\\_each\(\)](#) in that it's the simplest possible list iteration code, no prefetching is done. Use this for code that knows the list to be very short (empty or 1 entry) most of the time.

### 4.3.2.2 container\_of

```
#define container_of(  
    ptr,  
    type,  
    member )
```

**Value:**

```
((  
    const typeof( ((type *)0)->member ) *__mptr = (ptr); \br/>    (type *) ( (char *)__mptr - offsetof(type,member) );))
```

Casts a member of a structure out to the containing structure

**Parameters**

<i>ptr</i>	the pointer to the member.
<i>type</i>	the type of the container struct this is embedded in.
<i>member</i>	the name of the member within the struct.

### 4.3.2.3 hlist\_for\_each

```
#define hlist_for_each(  
    pos,  
    head )
```

**Value:**

```
for (pos = (head)->first; pos && ({ prefetch(pos->next); 1; }); \
    pos = pos->next)
```

**4.3.2.4 hlist\_for\_each\_entry**

```
#define hlist_for_each_entry(  
    tpos,  
    pos,  
    head,  
    member )
```

**Value:**

```
for (pos = (head)->first; \
    pos && ({ prefetch(pos->next); 1;}) && \
    ({ tpos = hlist_entry(pos, typeof(*tpos), member); 1;}); \
    pos = pos->next)
```

`hlist_for_each_entry` - iterate over list of given type `@tpos`: the type `*` to use as a loop counter. `@pos`: the `&struct hlist_node` to use as a loop counter. `@head`: the head for your list. `@member`: the name of the `hlist_node` within the struct.

**4.3.2.5 hlist\_for\_each\_entry\_continue**

```
#define hlist_for_each_entry_continue(  
    tpos,  
    pos,  
    member )
```

**Value:**

```
for (pos = (pos)->next; \
    pos && ({ prefetch(pos->next); 1;}) && \
    ({ tpos = hlist_entry(pos, typeof(*tpos), member); 1;}); \
    pos = pos->next)
```

`hlist_for_each_entry_continue` - iterate over a hlist continuing after existing point `@tpos`: the type `*` to use as a loop counter. `@pos`: the `&struct hlist_node` to use as a loop counter. `@member`: the name of the `hlist_node` within the struct.

**4.3.2.6 hlist\_for\_each\_entry\_from**

```
#define hlist_for_each_entry_from(  
    tpos,  
    pos,  
    member )
```

**Value:**

```
for (; pos && ({ prefetch(pos->next); 1;}) && \
    ({ tpos = hlist_entry(pos, typeof(*tpos), member); 1;}); \
    pos = pos->next)
```

`hlist_for_each_entry_from` - iterate over a hlist continuing from existing point `@tpos`: the type `*` to use as a loop counter. `@pos`: the `&struct hlist_node` to use as a loop counter. `@member`: the name of the `hlist_node` within the struct.

#### 4.3.2.7 hlist\_for\_each\_entry\_safe

```
#define hlist_for_each_entry_safe(
    tpos,
    pos,
    n,
    head,
    member )
```

**Value:**

```
for (pos = (head)->first; \
     pos && ({ n = pos->next; 1; }) && \
     ({ tpos = hlist_entry(pos, typeof(*tpos), member); 1; }); \
     pos = n)
```

hlist\_for\_each\_entry\_safe - iterate over list of given type safe against removal of list entry @tpos: the type \* to use as a loop counter. @pos: the &struct [hlist\\_node](#) to use as a loop counter.

: another &struct [hlist\\_node](#) to use as temporary storage @head: the head for your list. @member: the name of the [hlist\\_node](#) within the struct.

#### 4.3.2.8 hlist\_for\_each\_safe

```
#define hlist_for_each_safe(
    pos,
    n,
    head )
```

**Value:**

```
for (pos = (head)->first; pos && ({ n = pos->next; 1; }); \
     pos = n)
```

#### 4.3.2.9 INIT\_LIST\_HEAD

```
#define INIT_LIST_HEAD(
    ptr )
```

**Value:**

```
do { \
    (ptr)->next = (ptr); (ptr)->prev = (ptr); \
} while (0)
```

#### 4.3.2.10 list\_entry

```
#define list_entry(
    ptr,
    type,
    member ) container_of(ptr, type, member)
```

list\_entry - get the struct for this entry @ptr: the &struct [list\\_head](#) pointer. @type: the type of the struct this is embedded in. @member: the name of the list\_struct within the struct.

#### 4.3.2.11 list\_for\_each

```
#define list_for_each(
    pos,
    head )
```

**Value:**

```
for (pos = (head)->next; pos != (head); \
    pos = pos->next)
```

list\_for\_each - iterate over a list @pos: the &struct [list\\_head](#) to use as a loop counter. @head: the head for your list.

#### 4.3.2.12 list\_for\_each\_entry

```
#define list_for_each_entry(
    pos,
    head,
    member )
```

**Value:**

```
for (pos = list_entry((head)->next, typeof(*pos), member); \
    &pos->member != (head); \
    pos = list_entry(pos->member.next, typeof(*pos), member))
```

list\_for\_each\_entry - iterate over list of given type @pos: the type \* to use as a loop counter. @head: the head for your list. @member: the name of the list\_struct within the struct.

#### 4.3.2.13 list\_for\_each\_entry\_continue

```
#define list_for_each_entry_continue(
    pos,
    head,
    member )
```

**Value:**

```
for (pos = list_entry(pos->member.next, typeof(*pos), member); \
    &pos->member != (head); \
    pos = list_entry(pos->member.next, typeof(*pos), member))
```

list\_for\_each\_entry\_continue - iterate over list of given type continuing after existing point @pos: the type \* to use as a loop counter. @head: the head for your list. @member: the name of the list\_struct within the struct.

#### 4.3.2.14 list\_for\_each\_entry\_reverse

```
#define list_for_each_entry_reverse(
    pos,
    head,
    member )
```

**Value:**

```
for (pos = list_entry((head)->prev, typeof(*pos), member); \
    &pos->member != (head); \
    pos = list_entry(pos->member.prev, typeof(*pos), member))
```

list\_for\_each\_entry\_reverse - iterate backwards over list of given type. @pos: the type \* to use as a loop counter. @head: the head for your list. @member: the name of the list\_struct within the struct.

#### 4.3.2.15 list\_for\_each\_entry\_safe

```
#define list_for_each_entry_safe(  
    pos,  
    n,  
    head,  
    member )
```

**Value:**

```
for (pos = list_entry((head->next, typeof(*pos), member), \  
    n = list_entry(pos->member.next, typeof(*pos), member); \  
    &pos->member != (head); \  
    pos = n, n = list_entry(n->member.next, typeof(*n), member))
```

list\_for\_each\_entry\_safe - iterate over list of given type safe against removal of list entry @pos: the type \* to use as a loop counter.

: another type \* to use as temporary storage @head: the head for your list. @member: the name of the list\_struct within the struct.

#### 4.3.2.16 list\_for\_each\_entry\_safe\_continue

```
#define list_for_each_entry_safe_continue(  
    pos,  
    n,  
    head,  
    member )
```

**Value:**

```
for (pos = list_entry(pos->member.next, typeof(*pos), member), \  
    n = list_entry(pos->member.next, typeof(*pos), member); \  
    &pos->member != (head); \  
    pos = n, n = list_entry(n->member.next, typeof(*n), member))
```

list\_for\_each\_entry\_safe\_continue - iterate over list of given type continuing after existing point safe against removal of list entry @pos: the type \* to use as a loop counter.

: another type \* to use as temporary storage @head: the head for your list. @member: the name of the list\_struct within the struct.

#### 4.3.2.17 list\_for\_each\_entry\_safe\_reverse

```
#define list_for_each_entry_safe_reverse(  
    pos,  
    n,  
    head,  
    member )
```

**Value:**

```
for (pos = list_entry((head->prev, typeof(*pos), member), \  
    n = list_entry(pos->member.prev, typeof(*pos), member); \  
    &pos->member != (head); \  
    pos = n, n = list_entry(n->member.prev, typeof(*n), member))
```

list\_for\_each\_entry\_safe\_reverse - iterate backwards over list of given type safe against removal of list entry @pos: the type \* to use as a loop counter.

: another type \* to use as temporary storage @head: the head for your list. @member: the name of the list\_struct within the struct.

#### 4.3.2.18 list\_for\_each\_prev

```
#define list_for_each_prev(
    pos,
    head )
```

##### Value:

```
for (pos = (head)->prev; prefetch(pos->prev), pos != (head); \
    pos = pos->prev)
```

list\_for\_each\_prev - iterate over a list backwards @pos: the &struct [list\\_head](#) to use as a loop counter. @head: the head for your list.

#### 4.3.2.19 list\_for\_each\_safe

```
#define list_for_each_safe(
    pos,
    n,
    head )
```

##### Value:

```
for (pos = (head)->next, n = pos->next; pos != (head); \
    pos = n, n = pos->next)
```

list\_for\_each\_safe - iterate over a list safe against removal of list entry @pos: the &struct [list\\_head](#) to use as a loop counter.

: another &struct [list\\_head](#) to use as temporary storage @head: the head for your list.

#### 4.3.2.20 list\_prepare\_entry

```
#define list_prepare_entry(
    pos,
    head,
    member ) ((pos) ? : list\_entry(head, typeof(*pos), member))
```

list\_prepare\_entry - prepare a pos entry for use as a start point in list\_for\_each\_entry\_continue @pos: the type \* to use as a start point @head: the head of the list @member: the name of the list\_struct within the struct.

#### 4.3.2.21 offsetof

```
#define offsetof(
    TYPE,
    MEMBER ) ((size_t) &((TYPE *)0)->MEMBER)
```

Get offset of a member



## 4.4 list.h

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

```

1
32 #ifndef _LINUX_LIST_H
33 #define _LINUX_LIST_H
34
35 #include <stdio.h>
40
44 #define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
45
53 #define container_of(ptr, type, member) ({
54     const typeof( ((type *)0)->member ) *__mptr = (ptr);
55     (type *) ( (char *)__mptr - offsetof(type,member) );})
59 /*
60  * These are non-NULL pointers that will result in page faults
61  * under normal circumstances, used to verify that nobody uses
62  * non-initialized list entries.
63  */
64 #define LIST_POISON1  ((void *) 0x00100100)
65 #define LIST_POISON2  ((void *) 0x00200200)
66
76 struct list_head {
77     struct list_head *next, *prev;
78 };
79
80 #define LIST_HEAD_INIT(name) { &(name), &(name) }
81
82 #define LIST_HEAD(name) \
83     struct list_head name = LIST_HEAD_INIT(name)
84
85 #define INIT_LIST_HEAD(ptr) do { \
86     (ptr)->next = (ptr); (ptr)->prev = (ptr); \
87 } while (0)
88
89 /*
90  * Insert a new entry between two known consecutive entries.
91  *
92  * This is only for internal list manipulation where we know
93  * the prev/next entries already!
94  */
95 static inline void __list_add(struct list_head *new,
96                             struct list_head *prev,
97                             struct list_head *next)
98 {
99     next->prev = new;
100     new->next = next;
101     new->prev = prev;
102     prev->next = new;
103 }
104
113 static inline void list_add(struct list_head *new, struct list_head *head)
114 {
115     __list_add(new, head, head->next);
116 }
117
126 static inline void list_add_tail(struct list_head *new, struct list_head *head)
127 {
128     __list_add(new, head->prev, head);
129 }
130
131
132 /*
133  * Delete a list entry by making the prev/next entries
134  * point to each other.
135  *
136  * This is only for internal list manipulation where we know
137  * the prev/next entries already!
138  */
139 static inline void __list_del(struct list_head * prev, struct list_head * next)
140 {
141     next->prev = prev;
142     prev->next = next;
143 }
144
151 static inline void list_del(struct list_head *entry)
152 {
153     __list_del(entry->prev, entry->next);
154     entry->next = LIST_POISON1;
155     entry->prev = LIST_POISON2;
156 }
157
158
159
164 static inline void list_del_init(struct list_head *entry)

```

```

165 {
166     __list_del(entry->prev, entry->next);
167     INIT_LIST_HEAD(entry);
168 }
169
175 static inline void list_move(struct list_head *list, struct list_head *head)
176 {
177     __list_del(list->prev, list->next);
178     list_add(list, head);
179 }
180
186 static inline void list_move_tail(struct list_head *list,
187     struct list_head *head)
188 {
189     __list_del(list->prev, list->next);
190     list_add_tail(list, head);
191 }
192
197 static inline int list_empty(const struct list_head *head)
198 {
199     return head->next == head;
200 }
201
202 static inline void __list_splice(struct list_head *list,
203     struct list_head *head)
204 {
205     struct list_head *first = list->next;
206     struct list_head *last = list->prev;
207     struct list_head *at = head->next;
208
209     first->prev = head;
210     head->next = first;
211
212     last->next = at;
213     at->prev = last;
214 }
215
221 static inline void list_splice(struct list_head *list, struct list_head *head)
222 {
223     if (!list_empty(list))
224         __list_splice(list, head);
225 }
226
234 static inline void list_splice_init(struct list_head *list,
235     struct list_head *head)
236 {
237     if (!list_empty(list)) {
238         __list_splice(list, head);
239         INIT_LIST_HEAD(list);
240     }
241 }
242
249 #define list_entry(ptr, type, member) \
250     container_of(ptr, type, member)
251
258 #define list_for_each(pos, head) \
259     for (pos = (head)->next; pos != (head); \
260         pos = pos->next)
261
272 #define __list_for_each(pos, head) \
273     for (pos = (head)->next; pos != (head); pos = pos->next)
274
280 #define list_for_each_prev(pos, head) \
281     for (pos = (head)->prev; prefetch(pos->prev), pos != (head); \
282         pos = pos->prev)
283
290 #define list_for_each_safe(pos, n, head) \
291     for (pos = (head)->next, n = pos->next; pos != (head); \
292         pos = n, n = pos->next)
293
300 #define list_for_each_entry(pos, head, member) \
301     for (pos = list_entry((head)->next, typeof(*pos), member); \
302         &pos->member != (head); \
303         pos = list_entry(pos->member.next, typeof(*pos), member))
304
311 #define list_for_each_entry_reverse(pos, head, member) \
312     for (pos = list_entry((head)->prev, typeof(*pos), member); \
313         &pos->member != (head); \
314         pos = list_entry(pos->member.prev, typeof(*pos), member))
315
323 #define list_prepare_entry(pos, head, member) \
324     ((pos) ? : list_entry(head, typeof(*pos), member))
325
333 #define list_for_each_entry_continue(pos, head, member) \
334     for (pos = list_entry(pos->member.next, typeof(*pos), member); \
335         &pos->member != (head); \
336         pos = list_entry(pos->member.next, typeof(*pos), member))

```

```

337
345 #define list_for_each_entry_safe(pos, n, head, member)      \
346     for (pos = list_entry((head)->next, typeof(*pos), member), \
347         n = list_entry(pos->member.next, typeof(*pos), member); \
348         &pos->member != (head); \
349         pos = n, n = list_entry(n->member.next, typeof(*n), member))
350
359 #define list_for_each_entry_safe_continue(pos, n, head, member)      \
360     for (pos = list_entry(pos->member.next, typeof(*pos), member), \
361         n = list_entry(pos->member.next, typeof(*pos), member); \
362         &pos->member != (head); \
363         pos = n, n = list_entry(n->member.next, typeof(*n), member))
364
373 #define list_for_each_entry_safe_reverse(pos, n, head, member)      \
374     for (pos = list_entry((head)->prev, typeof(*pos), member), \
375         n = list_entry(pos->member.prev, typeof(*pos), member); \
376         &pos->member != (head); \
377         pos = n, n = list_entry(n->member.prev, typeof(*n), member))
378
379
380
381
382 /*
383  * Double linked lists with a single pointer list head.
384  * Mostly useful for hash tables where the two pointer list head is
385  * too wasteful.
386  * You lose the ability to access the tail in O(1).
387  */
388
389 struct hlist_head {
390     struct hlist_node *first;
391 };
392
393 struct hlist_node {
394     struct hlist_node *next, **pprev;
395 };
396
397 #define HLIST_HEAD_INIT { .first = NULL }
398 #define HLIST_HEAD(name) struct hlist_head name = { .first = NULL }
399 #define INIT_HLIST_HEAD(ptr) ((ptr)->first = NULL)
400 #define INIT_HLIST_NODE(ptr) ((ptr)->next = NULL, (ptr)->pprev = NULL)
401
402 static inline int hlist_unhashed(const struct hlist_node *h)
403 {
404     return !h->pprev;
405 }
406
407 static inline int hlist_empty(const struct hlist_head *h)
408 {
409     return !h->first;
410 }
411
412 static inline void __hlist_del(struct hlist_node *n)
413 {
414     struct hlist_node *next = n->next;
415     struct hlist_node **pprev = n->pprev;
416     *pprev = next;
417     if (next)
418         next->pprev = pprev;
419 }
420
421 static inline void hlist_del(struct hlist_node *n)
422 {
423     __hlist_del(n);
424     n->next = LIST_POISON1;
425     n->pprev = LIST_POISON2;
426 }
427
428
429 static inline void hlist_del_init(struct hlist_node *n)
430 {
431     if (n->pprev) {
432         __hlist_del(n);
433         INIT_HLIST_NODE(n);
434     }
435 }
436
437 static inline void hlist_add_head(struct hlist_node *n, struct hlist_head *h)
438 {
439     struct hlist_node *first = h->first;
440     n->next = first;
441     if (first)
442         first->pprev = &n->next;
443     h->first = n;
444     n->pprev = &h->first;
445 }
446

```

```

447
448
449 /* next must be != NULL */
450 static inline void hlist_add_before(struct hlist_node *n,
451                                   struct hlist_node *next)
452 {
453     n->pprev = next->pprev;
454     n->next = next;
455     next->pprev = &n->next;
456     *(n->pprev) = n;
457 }
458
459 static inline void hlist_add_after(struct hlist_node *n,
460                                   struct hlist_node *next)
461 {
462     next->next = n->next;
463     n->next = next;
464     next->pprev = &n->next;
465
466     if(next->next)
467         next->next->pprev = &next->next;
468 }
469
470
471
472 #define hlist_entry(ptr, type, member) container_of(ptr,type,member)
473
474 #define hlist_for_each(pos, head) \
475     for (pos = (head)->first; pos && ({ prefetch(pos->next); 1; }); \
476          pos = pos->next)
477
478 #define hlist_for_each_safe(pos, n, head) \
479     for (pos = (head)->first; pos && ({ n = pos->next; 1; }); \
480          pos = n)
481
482 #define hlist_for_each_entry(tpos, pos, head, member) \
483     for (pos = (head)->first; \
484          pos && ({ prefetch(pos->next); 1; }) && \
485          ({ tpos = hlist_entry(pos, typeof(*tpos), member); 1; }); \
486          pos = pos->next)
487
488 #define hlist_for_each_entry_continue(tpos, pos, member) \
489     for (pos = (pos)->next; \
490          pos && ({ prefetch(pos->next); 1; }) && \
491          ({ tpos = hlist_entry(pos, typeof(*tpos), member); 1; }); \
492          pos = pos->next)
493
494 #define hlist_for_each_entry_from(tpos, pos, member) \
495     for (; pos && ({ prefetch(pos->next); 1; }) && \
496          ({ tpos = hlist_entry(pos, typeof(*tpos), member); 1; }); \
497          pos = pos->next)
498
499 #define hlist_for_each_entry_safe(tpos, pos, n, head, member) \
500     for (pos = (head)->first; \
501          pos && ({ n = pos->next; 1; }) && \
502          ({ tpos = hlist_entry(pos, typeof(*tpos), member); 1; }); \
503          pos = n)
504
505 #endif

```

## 4.5 ums\_lib.c File Reference

Contains implementations of the essential UMS library functions.

```

#include "ums_lib.h"
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <pthread.h>
#include <fcntl.h>
#include <errno.h>
#include <sched.h>
#include <unistd.h>

```

## Macros

- `#define _GNU_SOURCE`
- `#define create_list_params(size) (list_params_t*)malloc(sizeof(list_params_t) + size * sizeof(ums_wid_t))`

## Functions

- `int open_device ()`  
*Opens UMS device Uses mutex to protect a shared resource from simultaneous access by multiple threads.*
- `int close_device ()`  
*Closes UMS device Uses mutex to protect a shared resource from simultaneous access by multiple threads.*
- `int ums_enter ()`  
*Requests UMS kernel module to manage current process*
- `int ums_exit ()`  
*Requests UMS kernel module to finish management of the current process*
- `ums_clid_t ums_create_completion_list ()`  
*Requests UMS kernel module to create a completion lists*
- `ums_wid_t ums_create_worker_thread (ums_clid_t clid, unsigned long stack_size, void(*entry_point)(void *), void *args)`  
*Requests UMS kernel module to create a worker thread assigned to specific completion list Library requests UMS kernel module to create a worker thread by passing worker\_params.*
- `ums_sid_t ums_create_scheduler (ums_clid_t clid, void(*entry_point)())`  
*Wrapper function that creates pthreads which eventually request UMS kernel module to create a scheduler UMS library uses pthread library to create process threads that will become scheduler threads. Each pthread jumps to ums\_enter\_scheduling\_mode() function and requests UMS kernel module to create a scheduler by passing scheduler\_params. After succesful creation of the scheduler by the UMS kernel module, created pthread becomes scheduler. It starts scheduler work by jumping to the entry point assigned by a user and stays there until ums\_exit\_scheduling\_mode() is called. Here list\_params is also created for the future calls of ums\_dequeue\_completion\_list\_items() by a scheduler (since in this stage the completion list has been fully populated and cannot be modified later).*
- `void * ums_enter_scheduling_mode (void *args)`  
*Actual function that is called by a pthread to request the UMS kernel module in order create a scheduler and assign a completion list to it Additionally assigns a CPU core on which the scheduler will operate based on available cores.*
- `int ums_exit_scheduling_mode ()`  
*Called by a scheduler to signal the UMS kernel module about the completion of scheduling mode Restores instruction, stack and base pointers to return back to ums\_enter\_scheduling\_mode() function to perform pthread\_exit()*
- `int ums_execute_thread (ums_wid_t wid)`  
*Called by a scheduler to request UMS kernel module to execute a worker thread with specific ID*
- `int ums_thread_yield (worker_status_t status)`  
*Called by a worker thread to pause or complete the execution Depending on the value of the argument, the function will:*
- `int ums_thread_pause ()`  
*Called by a worker thread to pause the execution Wrapper that calls ums\_thread\_yield() with an argument PAUSE.*
- `int ums_thread_exit ()`  
*Called by a worker thread to complete the execution Wrapper that calls ums\_thread\_yield() with an argument FINISH.*
- `list_params_t * ums_dequeue_completion_list_items ()`  
*Called by a scheduler to request UMS kernel module to provide a list of available worker threads that can be scheduled The function passes a global list\_params from the ums\_completion\_list\_node structure to the UMS kernel module The kernel module populates the structure with the list of available workers and sets the number of those available workers. Each scheduler has own copy of the list\_params, but can notify other scheduler about the state of the completion list (if shared) by updating its' state. Thus other schedulers do not have to perform ioctl call, just update their own list\_params and set its' state to FINISHED.*
- `ums_wid_t ums_get_next_worker_thread (list_params_t *list)`

Called by a scheduler, after performing `ums_dequeue_completion_list_items()`, to find a next available worker thread from the completion list. This function always has to be run after calling `ums_dequeue_completion_list_items()`, since it will populate the list in the correct way to be processed. Passing a manually created list parameter will result in undefined behaviour.

- `int cleanup ()`  
Performs a cleanup by deleting all the data structures allocated by the library
- `ums_completion_list_node_t * check_if_completion_list_exists (ums_clid_t clid)`  
Checks if the completion list with a passed ID exists or not
- `ums_worker_t * check_if_worker_exists (ums_wid_t wid)`  
Checks if the worker thread with a passed ID exists or not
- `ums_scheduler_t * check_if_scheduler_exists ()`  
Checks if the scheduler for the current pthread exists or not
- `__attribute__((constructor))`
- `__attribute__((destructor))`

## Variables

- `int ums_dev = -UMS_ERROR`
- `pthread_mutex_t ums_mutex = PTHREAD_MUTEX_INITIALIZER`
- `ums_completion_list_t completion_lists`
- `ums_worker_list_t workers`
- `ums_scheduler_list_t schedulers`
- `__thread ums_clid_t completion_list_id`

### 4.5.1 Detailed Description

Contains implementations of the essential UMS library functions.

Copyright (C) 2021 Bektur Umarbaev [hrafnulf13@gmail.com](mailto:hrafnulf13@gmail.com)

This file is part of the User Mode thread Scheduling (UMS) library.

UMS library is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

UMS library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with UMS library. If not, see <http://www.gnu.org/licenses/>.

#### Author

Bektur Umarbaev [hrafnulf13@gmail.com](mailto:hrafnulf13@gmail.com)

#### Date

## 4.5.2 Function Documentation

### 4.5.2.1 check\_if\_completion\_list\_exists()

```
ums_completion_list_node_t * check_if_completion_list_exists (  
    ums_clid_t clid )
```

Checks if the completion list with a passed ID exists or not

**Parameters**

<i>clid</i>	Completion list ID
-------------	--------------------

**Returns**

returns a pointer to the existing completion list structure if it exists, NULL otherwise

**4.5.2.2 check\_if\_scheduler\_exists()**

```
ums_scheduler_t * check_if_scheduler_exists ( )
```

Checks if the scheduler for the current pthread exists or not

**Returns**

returns a pointer to the existing scheduler structure if it exists, NULL otherwise

**4.5.2.3 check\_if\_worker\_exists()**

```
ums_worker_t * check_if_worker_exists (
    ums_wid_t wid )
```

Checks if the worker thread with a passed ID exists or not

**Parameters**

<i>wid</i>	Worker thread ID
------------	------------------

**Returns**

returns a UMS\_SUCCESS if worker thread exists, UMS\_ERROR\_WORKER\_NOT\_FOUND otherwise

**4.5.2.4 cleanup()**

```
int cleanup ( )
```

Performs a cleanup by deleting all the data structures allocated by the library

**Returns**

returns UMS\_SUCCESS when succesful or UMS\_ERROR if there are any errors



#### 4.5.2.5 close\_device()

```
int close_device ( )
```

Closes UMS device Uses mutex to protect a shared resource from simultaneous access by multiple threads.

##### Returns

returns UMS\_SUCCESS when succesful or UMS\_ERROR if there are any errors

#### 4.5.2.6 open\_device()

```
int open_device ( )
```

Opens UMS device Uses mutex to protect a shared resource from simultaneous access by multiple threads.

##### Returns

returns UMS\_SUCCESS when succesful or UMS\_ERROR if there are any errors

#### 4.5.2.7 ums\_create\_completion\_list()

```
ums_clid_t ums_create_completion_list ( )
```

Requests UMS kernel module to create a completion lists

##### Returns

returns Completion list ID

#### 4.5.2.8 ums\_create\_scheduler()

```
ums_sid_t ums_create_scheduler (
    ums_clid_t clid,
    void(*)() entry_point )
```

Wrapper function that creates pthreads which eventually request UMS kernel module to create a scheduler UMS library uses pthread library to create process threads that will become scheduler threads. Each pthread jumps to [ums\\_enter\\_scheduling\\_mode\(\)](#) function and requests UMS kernel module to create a scheduler by passing [scheduler\\_params](#). After succesful creation of the scheduler by the UMS kernel module, created pthread becomes scheduler. It starts scheduler work by jumping to the entry point assigned by a user and stays there until [ums\\_exit\\_scheduling\\_mode\(\)](#) is called. Here [list\\_params](#) is also created for the future calls of [ums\\_dequeue\\_completion\\_list\\_items\(\)](#) by a scheduler (since in this stage the completion list has been fully populated and cannot be modified later).

**Parameters**

<i>clid</i>	ID of the completion list that is assigned to the scheduler
<i>entry_point</i>	Function pointer and an entry point set by a user, that serves as a starting point of the scheduler. It is a scheduling function that determines the next thread to be scheduled

**Returns**

returns Scheduler ID

**4.5.2.9 ums\_create\_worker\_thread()**

```
ums_wid_t ums_create_worker_thread (
    ums_clid_t clid,
    unsigned long stack_size,
    void(*) (void *) entry_point,
    void * args )
```

Requests UMS kernel module to create a worker thread assigned to specific completion list Library requests UMS kernel module to create a worker thread by passing [worker\\_params](#).

**Parameters**

<i>clid</i>	ID of the completion list where worker thread is assigned to
<i>stack_size</i>	Stack size of the worker thread set by a user
<i>entry_point</i>	Function pointer and an entry point set by a user, that serves as a starting point of the worker thread
<i>args</i>	Pointer of the function arguments that are passed to the entry point/function

**Returns**

returns Worker ID

**4.5.2.10 ums\_dequeue\_completion\_list\_items()**

```
list_params_t * ums_dequeue_completion_list_items ( )
```

Called by a scheduler to request UMS kernel module to provide a list of available worker threads that can be scheduled The function passes a global [list\\_params](#) from the [ums\\_completion\\_list\\_node](#) structure to the UMS kernel module The kernel module populates the structure with the list of available workers and sets the number of those available workers. Each scheduler has own copy of the [list\\_params](#), but can notify other scheduler about the state of the completion list (if shared) by updating its' state. Thus other schedulers do not have to perform ioctl call, just update their own [list\\_params](#) and set its' [state](#) to `FINISHED`.

**Returns**

returns the pointer to a shared [list\\_params](#) structure which contains an array of available workers that can be scheduled

#### 4.5.2.11 ums\_enter()

```
int ums_enter ( )
```

Requests UMS kernel module to manage current process

##### Returns

returns UMS\_SUCCESS when succesful or UMS\_ERROR if there are any errors

#### 4.5.2.12 ums\_enter\_scheduling\_mode()

```
void * ums_enter_scheduling_mode (
    void * args )
```

Actual function that is called by a pthread to request the UMS kernel module in order create a scheduler and assign a completion list to it Additionally assigns a CPU core on which the scheduler will operate based on available cores.

##### Parameters

<i>args</i>	Pointer to <a href="#">scheduler_params</a> that is passed in order to create a scheduler
-------------	---

##### Returns

#### 4.5.2.13 ums\_execute\_thread()

```
int ums_execute_thread (
    ums_wid_t wid )
```

Called by a scheduler to request UMS kernel module to execute a worker thread with specific ID

##### Parameters

<i>wid</i>	ID of the worker thread that to be executed
------------	---

##### Returns

#### 4.5.2.14 ums\_exit()

```
int ums_exit ( )
```

Requests UMS kernel module to finish management of the current process

##### Returns

returns UMS\_SUCCESS when succesful or UMS\_ERROR if there are any errors

#### 4.5.2.15 ums\_exit\_scheduling\_mode()

```
int ums_exit_scheduling_mode ( )
```

Called by a scheduler to signal the UMS kernel module about the completion of scheduling mode Restores instruction, stack and base pointers to return back to [ums\\_enter\\_scheduling\\_mode\(\)](#) function to perform pthread\_exit()

##### Returns

#### 4.5.2.16 ums\_get\_next\_worker\_thread()

```
ums_wid_t ums_get_next_worker_thread (
    list_params_t * list )
```

Called by a scheduler, after performing [ums\\_dequeue\\_completion\\_list\\_items\(\)](#), to find a next available worker thread from the completion list This function always has to be run after calling [ums\\_dequeue\\_completion\\_list\\_items\(\)](#), since it will populate the list in the correct way to be processed Passing a manually created list parameter will result in undefined behaviour.

##### Parameters

<i>list</i>	List parameter that is created after <a href="#">ums_dequeue_completion_list_items()</a> call and contains the list of available workers
-------------	--

##### Returns

returns a next available worker thread that can be scheduled, or error values otherwise

#### 4.5.2.17 ums\_thread\_exit()

```
int ums_thread_exit ( )
```

Called by a worker thread to complete the execution Wrapper that calls [ums\\_thread\\_yield\(\)](#) with an argument FINISH.

## Returns

### 4.5.2.18 ums\_thread\_pause()

```
int ums_thread_pause ( )
```

Called by a worker thread to pause the execution Wrapper that calls [ums\\_thread\\_yield\(\)](#) with an argument `PAUSE`.

## Returns

### 4.5.2.19 ums\_thread\_yield()

```
int ums_thread_yield (
    worker_status_t status )
```

Called by a worker thread to pause or complete the execution Depending on the value of the argument, the function will:

- Remove the worker thread from the list of worker threads that can be scheduled, thus completes the execution;
- Push it back to the list of available worker thread, thus pauses its' execution and can be rescheduled later.

## Parameters

<i>status</i>	defines the status of the execution flow of the worker thread (passing <code>PAUSE</code> will pause the execution, when <code>FINISH</code> will complete it)
---------------	--

## Returns

## 4.5.3 Variable Documentation

### 4.5.3.1 completion\_lists

```
ums_completion_list_t completion_lists
```

**Initial value:**

```
= {
    .list = LIST_HEAD_INIT(completion_lists.list),
    .count = 0
}
```

**4.5.3.2 schedulers**

```
ums_scheduler_list_t schedulers
```

**Initial value:**

```
= {
    .list = LIST_HEAD_INIT(schedulers.list),
    .count = 0
}
```

**4.5.3.3 workers**

```
ums_worker_list_t workers
```

**Initial value:**

```
= {
    .list = LIST_HEAD_INIT(workers.list),
    .count = 0
}
```

**4.6 ums\_lib.h File Reference**

The header that contains essential UMS library functions and has to be included by the user in order to use the UMS library.

```
#include "const.h"
#include "list.h"
#include <pthread.h>
```

**Data Structures**

- struct [ums\\_completion\\_list](#)  
*The list of the completion lists created by the process*
- struct [ums\\_completion\\_list\\_node](#)  
*Represents a node in the [ums\\_completion\\_list](#)*
- struct [ums\\_worker\\_list](#)  
*The list of the worker threads created by the process*
- struct [ums\\_worker](#)  
*Represents a node in the [ums\\_worker\\_list](#)*
- struct [ums\\_scheduler\\_list](#)  
*The list of the schedulers created by the process*
- struct [ums\\_scheduler](#)  
*Represents a node in the [ums\\_scheduler\\_list](#)*

## Macros

- `#define UMS_DEVICE "/dev/ums"`
- `#define init(type) (type*)malloc(sizeof(type))`
- `#define delete(val) free(val)`

## Typedefs

- typedef struct [ums\\_completion\\_list](#) **ums\_completion\_list\_t**  
*The list of the completion lists created by the process*
- typedef struct [ums\\_completion\\_list\\_node](#) **ums\_completion\_list\_node\_t**  
*Represents a node in the [ums\\_completion\\_list](#)*
- typedef struct [ums\\_worker](#) **ums\_worker\_t**  
*Represents a node in the [ums\\_worker\\_list](#)*
- typedef struct [ums\\_worker\\_list](#) **ums\_worker\_list\_t**  
*The list of the worker threads created by the process*
- typedef struct [ums\\_scheduler\\_list](#) **ums\_scheduler\_list\_t**  
*The list of the schedulers created by the process*
- typedef struct [ums\\_scheduler](#) **ums\_scheduler\_t**  
*Represents a node in the [ums\\_scheduler\\_list](#)*

## Functions

- int [ums\\_enter](#) ()  
*Requests UMS kernel module to manage current process*
- int [ums\\_exit](#) ()  
*Requests UMS kernel module to finish management of the current process*
- [ums\\_clid\\_t](#) [ums\\_create\\_completion\\_list](#) ()  
*Requests UMS kernel module to create a completion lists*
- [ums\\_wid\\_t](#) [ums\\_create\\_worker\\_thread](#) ([ums\\_clid\\_t](#) clid, unsigned long stack\_size, void(\*entry\_point)(void \*), void \*args)  
*Requests UMS kernel module to create a worker thread assigned to specific completion list Library requests UMS kernel module to create a worker thread by passing [worker\\_params](#).*
- [ums\\_sid\\_t](#) [ums\\_create\\_scheduler](#) ([ums\\_clid\\_t](#) clid, void(\*entry\_point)(void \*))
- void \* [ums\\_enter\\_scheduling\\_mode](#) (void \*args)  
*Actual function that is called by a pthread to request the UMS kernel module in order create a scheduler and assign a completion list to it Additionally assigns a CPU core on which the scheduler will operate based on available cores.*
- int [ums\\_exit\\_scheduling\\_mode](#) ()  
*Called by a scheduler to signal the UMS kernel module about the completion of scheduling mode Restores instruction, stack and base pointers to return back to [ums\\_enter\\_scheduling\\_mode\(\)](#) function to perform pthread\_exit()*
- int [ums\\_execute\\_thread](#) ([ums\\_wid\\_t](#) wid)  
*Called by a scheduler to request UMS kernel module to execute a worker thread with specific ID*
- int [ums\\_thread\\_yield](#) ()
- int [ums\\_thread\\_pause](#) ()  
*Called by a worker thread to pause the execution Wrapper that calls [ums\\_thread\\_yield\(\)](#) with an argument PAUSE.*
- int [ums\\_thread\\_exit](#) ()  
*Called by a worker thread to complete the execution Wrapper that calls [ums\\_thread\\_yield\(\)](#) with an argument FINISH.*
- [list\\_params\\_t](#) \* [ums\\_dequeue\\_completion\\_list\\_items](#) ()

Called by a scheduler to request UMS kernel module to provide a list of available worker threads that can be scheduled. The function passes a global `list_params` from the `ums_completion_list_node` structure to the UMS kernel module. The kernel module populates the structure with the list of available workers and sets the number of those available workers. Each scheduler has own copy of the `list_params`, but can notify other scheduler about the state of the completion list (if shared) by updating its' state. Thus other schedulers do not have to perform `ioctl` call, just update their own `list_params` and set its' `state` to `FINISHED`.

- `ums_wid_t ums_get_next_worker_thread(list_params_t *list)`

Called by a scheduler, after performing `ums_dequeue_completion_list_items()`, to find a next available worker thread from the completion list. This function always has to be run after calling `ums_dequeue_completion_list_items()`, since it will populate the list in the correct way to be processed. Passing a manually created list parameter will result in undefined behaviour.

- `int open_device()`

Opens UMS device. Uses mutex to protect a shared resource from simultaneous access by multiple threads.

- `int close_device()`

Closes UMS device. Uses mutex to protect a shared resource from simultaneous access by multiple threads.

- `int cleanup()`

Performs a cleanup by deleting all the data structures allocated by the library.

- `ums_completion_list_node_t * check_if_completion_list_exists(ums_clid_t clid)`

Checks if the completion list with a passed ID exists or not.

- `ums_worker_t * check_if_worker_exists(ums_wid_t wid)`

Checks if the worker thread with a passed ID exists or not.

- `ums_scheduler_t * check_if_scheduler_exists()`

Checks if the scheduler for the current pthread exists or not.

### 4.6.1 Detailed Description

The header that contains essential UMS library functions and has to be included by the user in order to use the UMS library.

Copyright (C) 2021 Bektur Umarbaev [hrafnulf13@gmail.com](mailto:hrafnulf13@gmail.com)

This file is part of the User Mode thread Scheduling (UMS) library.

UMS library is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

UMS library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with UMS library. If not, see <http://www.gnu.org/licenses/>.

#### Author

Bektur Umarbaev [hrafnulf13@gmail.com](mailto:hrafnulf13@gmail.com)

#### Date



## 4.6.2 Function Documentation

### 4.6.2.1 check\_if\_completion\_list\_exists()

```
ums_completion_list_node_t * check_if_completion_list_exists (
    ums_clid_t clid )
```

Checks if the completion list with a passed ID exists or not

#### Parameters

<i>clid</i>	Completion list ID
-------------	--------------------

#### Returns

returns a pointer to the existing completion list structure if it exists, NULL otherwise

### 4.6.2.2 check\_if\_scheduler\_exists()

```
ums_scheduler_t * check_if_scheduler_exists ( )
```

Checks if the scheduler for the current pthread exists or not

#### Returns

returns a pointer to the existing scheduler structure if it exists, NULL otherwise

### 4.6.2.3 check\_if\_worker\_exists()

```
ums_worker_t * check_if_worker_exists (
    ums_wid_t wid )
```

Checks if the worker thread with a passed ID exists or not

#### Parameters

<i>wid</i>	Worker thread ID
------------	------------------

#### Returns

returns a UMS\_SUCCESS if worker thread exists, UMS\_ERROR\_WORKER\_NOT\_FOUND otherwise

#### 4.6.2.4 cleanup()

```
int cleanup ( )
```

Performs a cleanup by deleting all the data structures allocated by the library

##### Returns

returns UMS\_SUCCESS when succesful or UMS\_ERROR if there are any errors

#### 4.6.2.5 close\_device()

```
int close_device ( )
```

Closes UMS device Uses mutex to protect a shared resource from simultaneous access by multiple threads.

##### Returns

returns UMS\_SUCCESS when succesful or UMS\_ERROR if there are any errors

#### 4.6.2.6 open\_device()

```
int open_device ( )
```

Opens UMS device Uses mutex to protect a shared resource from simultaneous access by multiple threads.

##### Returns

returns UMS\_SUCCESS when succesful or UMS\_ERROR if there are any errors

#### 4.6.2.7 ums\_create\_completion\_list()

```
ums_clid_t ums_create_completion_list ( )
```

Requests UMS kernel module to create a completion lists

##### Returns

returns Completion list ID

#### 4.6.2.8 ums\_create\_worker\_thread()

```
ums_wid_t ums_create_worker_thread (
    ums_clid_t clid,
    unsigned long stack_size,
    void(*) (void *) entry_point,
    void * args )
```

Requests UMS kernel module to create a worker thread assigned to specific comletion list Library requests UMS kernel module to create a worker thread by passing [worker\\_params](#).

## Parameters

<i>clid</i>	ID of the completion list where worker thread is assigned to
<i>stack_size</i>	Stack size of the worker thread set by a user
<i>entry_point</i>	Function pointer and an entry point set by a user, that serves as a starting point of the worker thread
<i>args</i>	Pointer of the function arguments that are passed to the entry point/function

## Returns

returns Worker ID

#### 4.6.2.9 ums\_dequeue\_completion\_list\_items()

```
list_params_t * ums_dequeue_completion_list_items ( )
```

Called by a scheduler to request UMS kernel module to provide a list of available worker threads that can be scheduled. The function passes a global [list\\_params](#) from the [ums\\_completion\\_list\\_node](#) structure to the UMS kernel module. The kernel module populates the structure with the list of available workers and sets the number of those available workers. Each scheduler has its own copy of the [list\\_params](#), but can notify other scheduler about the state of the completion list (if shared) by updating its' state. Thus other schedulers do not have to perform ioctl call, just update their own [list\\_params](#) and set its' [state](#) to `FINISHED`.

## Returns

returns the pointer to a shared [list\\_params](#) structure which contains an array of available workers that can be scheduled

#### 4.6.2.10 ums\_enter()

```
int ums_enter ( )
```

Requests UMS kernel module to manage current process

## Returns

returns `UMS_SUCCESS` when successful or `UMS_ERROR` if there are any errors

#### 4.6.2.11 ums\_enter\_scheduling\_mode()

```
void * ums_enter_scheduling_mode (
    void * args )
```

Actual function that is called by a pthread to request the UMS kernel module in order to create a scheduler and assign a completion list to it. Additionally, it assigns a CPU core on which the scheduler will operate based on available cores.

**Parameters**

<i>args</i>	Pointer to <a href="#">scheduler_params</a> that is passed in order to create a scheduler
-------------	---

**Returns****4.6.2.12 ums\_execute\_thread()**

```
int ums_execute_thread (
    ums_wid_t wid )
```

Called by a scheduler to request UMS kernel module to execute a worker thread with specific ID

**Parameters**

<i>wid</i>	ID of the worker thread that to be executed
------------	---

**Returns****4.6.2.13 ums\_exit()**

```
int ums_exit ( )
```

Requests UMS kernel module to finish management of the current process

**Returns**

returns UMS\_SUCCESS when succesful or UMS\_ERROR if there are any errors

**4.6.2.14 ums\_exit\_scheduling\_mode()**

```
int ums_exit_scheduling_mode ( )
```

Called by a scheduler to signal the UMS kernel module about the completion of scheduling mode Restores instruction, stack and base pointers to return back to [ums\\_enter\\_scheduling\\_mode\(\)](#) function to perform pthread\_exit()

**Returns**

#### 4.6.2.15 ums\_get\_next\_worker\_thread()

```
ums_wid_t ums_get_next_worker_thread (
    list_params_t * list )
```

Called by a scheduler, after performing [ums\\_dequeue\\_completion\\_list\\_items\(\)](#), to find a next available worker thread from the completion list. This function always has to be run after calling [ums\\_dequeue\\_completion\\_list\\_items\(\)](#), since it will populate the list in the correct way to be processed. Passing a manually created list parameter will result in undefined behaviour.

##### Parameters

<i>list</i>	List parameter that is created after <a href="#">ums_dequeue_completion_list_items()</a> call and contains the list of available workers
-------------	--

##### Returns

returns a next available worker thread that can be scheduled, or error values otherwise

#### 4.6.2.16 ums\_thread\_exit()

```
int ums_thread_exit ( )
```

Called by a worker thread to complete the execution Wrapper that calls [ums\\_thread\\_yield\(\)](#) with an argument `FINISH`.

##### Returns

#### 4.6.2.17 ums\_thread\_pause()

```
int ums_thread_pause ( )
```

Called by a worker thread to pause the execution Wrapper that calls [ums\\_thread\\_yield\(\)](#) with an argument `PAUSE`.

##### Returns

## 4.7 ums\_lib.h

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

```

1
29 #pragma once
30
31 #include "const.h"
32 #include "list.h"
33 #include <pthread.h>
34
35 #define UMS_DEVICE "/dev/ums"
36
37 typedef struct ums_completion_list ums_completion_list_t;
38 typedef struct ums_completion_list_node ums_completion_list_node_t;
39 typedef struct ums_worker ums_worker_t;
40 typedef struct ums_worker_list ums_worker_list_t;
41 typedef struct ums_scheduler_list ums_scheduler_list_t;
42 typedef struct ums_scheduler ums_scheduler_t;
43
44 int ums_enter();
45 int ums_exit();
46
47 ums_clid_t ums_create_completion_list();
48 ums_wid_t ums_create_worker_thread(ums_clid_t clid, unsigned long stack_size, void (*entry_point)(void
    *), void *args);
49 ums_sid_t ums_create_scheduler(ums_clid_t clid, void (*entry_point)(void *));
50 void *ums_enter_scheduling_mode(void *args);
51 int ums_exit_scheduling_mode();
52 int ums_execute_thread(ums_wid_t wid);
53 int ums_thread_yield();
54 int ums_thread_pause();
55 int ums_thread_exit();
56 list_params_t *ums_dequeue_completion_list_items();
57 ums_wid_t ums_get_next_worker_thread(list_params_t *list);
58
59 int open_device();
60 int close_device();
61 int cleanup();
62 ums_completion_list_node_t *check_if_completion_list_exists(ums_clid_t clid);
63 ums_worker_t *check_if_worker_exists(ums_wid_t wid);
64 ums_scheduler_t *check_if_scheduler_exists();
65
66 typedef struct ums_completion_list {
67     struct list_head list;
68     unsigned int count;
69 } ums_completion_list_t;
70
71 typedef struct ums_completion_list_node {
72     ums_clid_t clid;
73     state_t state;
74     unsigned int worker_count;
75     struct list_head list;
76     list_params_t *list_params;
77 } ums_completion_list_node_t;
78
79 typedef struct ums_worker_list {
80     struct list_head list;
81     unsigned int count;
82 } ums_worker_list_t;
83
84 typedef struct ums_worker {
85     ums_wid_t wid;
86     state_t state;
87     struct list_head list;
88     worker_params_t *worker_params;
89 } ums_worker_t;
90
91 typedef struct ums_scheduler_list {
92     struct list_head list;
93     unsigned int count;
94 } ums_scheduler_list_t;
95
96 typedef struct ums_scheduler {
97     struct list_head list;
98     pthread_t tid;
99     ums_wid_t wid;
100     scheduler_params_t *sched_params;
101     list_params_t *list_params;
102 } ums_scheduler_t;
103
104 #define init(type) (type*)malloc(sizeof(type))
105 #define delete(val) free(val)
106 #define create_list_params(size) (list_params_t*)malloc(sizeof(list_params_t) + size *
    sizeof(ums_wid_t))

```

# Index

- [\\_\\_list\\_for\\_each](#)  
[list.h, 21](#)
- [check\\_if\\_completion\\_list\\_exists](#)
  - [ums\\_lib.c, 33](#)
  - [ums\\_lib.h, 43](#)
- [check\\_if\\_scheduler\\_exists](#)
  - [ums\\_lib.c, 34](#)
  - [ums\\_lib.h, 43](#)
- [check\\_if\\_worker\\_exists](#)
  - [ums\\_lib.c, 34](#)
  - [ums\\_lib.h, 43](#)
- [cleanup](#)
  - [ums\\_lib.c, 34](#)
  - [ums\\_lib.h, 44](#)
- [clid](#)
  - [scheduler\\_params, 7](#)
  - [ums\\_completion\\_list\\_node, 9](#)
  - [worker\\_params, 14](#)
- [close\\_device](#)
  - [ums\\_lib.c, 34](#)
  - [ums\\_lib.h, 44](#)
- [completion\\_lists](#)
  - [ums\\_lib.c, 39](#)
- [const.h, 15](#)
  - [FINISH, 18](#)
  - [FINISHED, 18](#)
  - [IDLE, 18](#)
  - [PAUSE, 18](#)
  - [RUNNING, 18](#)
  - [state, 17](#)
  - [worker\\_status, 18](#)
- [container\\_of](#)
  - [list.h, 21](#)
- [core\\_id](#)
  - [scheduler\\_params, 7](#)
- [count](#)
  - [ums\\_completion\\_list, 8](#)
  - [ums\\_scheduler\\_list, 11](#)
  - [ums\\_worker\\_list, 13](#)
- [entry\\_point](#)
  - [scheduler\\_params, 8](#)
  - [worker\\_params, 14](#)
- [FINISH](#)
  - [const.h, 18](#)
- [FINISHED](#)
  - [const.h, 18](#)
- [function\\_args](#)
  - [worker\\_params, 14](#)
- [hlist\\_for\\_each](#)
  - [list.h, 21](#)
- [hlist\\_for\\_each\\_entry](#)
  - [list.h, 22](#)
- [hlist\\_for\\_each\\_entry\\_continue](#)
  - [list.h, 22](#)
- [hlist\\_for\\_each\\_entry\\_from](#)
  - [list.h, 22](#)
- [hlist\\_for\\_each\\_entry\\_safe](#)
  - [list.h, 22](#)
- [hlist\\_for\\_each\\_safe](#)
  - [list.h, 23](#)
- [hlist\\_head, 5](#)
- [hlist\\_node, 5](#)
- [IDLE](#)
  - [const.h, 18](#)
- [INIT\\_LIST\\_HEAD](#)
  - [list.h, 23](#)
- [list.h, 19](#)
  - [\\_\\_list\\_for\\_each, 21](#)
  - [container\\_of, 21](#)
  - [hlist\\_for\\_each, 21](#)
  - [hlist\\_for\\_each\\_entry, 22](#)
  - [hlist\\_for\\_each\\_entry\\_continue, 22](#)
  - [hlist\\_for\\_each\\_entry\\_from, 22](#)
  - [hlist\\_for\\_each\\_entry\\_safe, 22](#)
  - [hlist\\_for\\_each\\_safe, 23](#)
  - [INIT\\_LIST\\_HEAD, 23](#)
  - [list\\_entry, 23](#)
  - [list\\_for\\_each, 23](#)
  - [list\\_for\\_each\\_entry, 24](#)
  - [list\\_for\\_each\\_entry\\_continue, 24](#)
  - [list\\_for\\_each\\_entry\\_reverse, 24](#)
  - [list\\_for\\_each\\_entry\\_safe, 24](#)
  - [list\\_for\\_each\\_entry\\_safe\\_continue, 25](#)
  - [list\\_for\\_each\\_entry\\_safe\\_reverse, 25](#)
  - [list\\_for\\_each\\_prev, 25](#)
  - [list\\_for\\_each\\_safe, 26](#)
  - [list\\_prepare\\_entry, 26](#)
  - [offsetof, 26](#)
- [list\\_entry](#)
  - [list.h, 23](#)
- [list\\_for\\_each](#)
  - [list.h, 23](#)
- [list\\_for\\_each\\_entry](#)
  - [list.h, 24](#)

- list\_for\_each\_entry\_continue
  - list.h, 24
- list\_for\_each\_entry\_reverse
  - list.h, 24
- list\_for\_each\_entry\_safe
  - list.h, 24
- list\_for\_each\_entry\_safe\_continue
  - list.h, 25
- list\_for\_each\_entry\_safe\_reverse
  - list.h, 25
- list\_for\_each\_prev
  - list.h, 25
- list\_for\_each\_safe
  - list.h, 26
- list\_head, 5
- list\_params, 6
  - size, 6
  - state, 6
  - ums\_completion\_list\_node, 9
  - ums\_scheduler, 10
  - worker\_count, 6
  - workers, 7
- list\_prepare\_entry
  - list.h, 26
- offsetof
  - list.h, 26
- open\_device
  - ums\_lib.c, 35
  - ums\_lib.h, 44
- PAUSE
  - const.h, 18
- RUNNING
  - const.h, 18
- sched\_params
  - ums\_scheduler, 10
- scheduler\_params, 7
  - clid, 7
  - core\_id, 7
  - entry\_point, 8
  - sid, 8
- schedulers
  - ums\_lib.c, 40
- sid
  - scheduler\_params, 8
- size
  - list\_params, 6
- stack\_addr
  - worker\_params, 14
- stack\_size
  - worker\_params, 14
- state
  - const.h, 17
  - list\_params, 6
  - ums\_completion\_list\_node, 9
  - ums\_worker, 12
- tid
  - ums\_scheduler, 11
- ums\_completion\_list, 8
  - count, 8
- ums\_completion\_list\_node, 9
  - clid, 9
  - list\_params, 9
  - state, 9
  - worker\_count, 10
- ums\_create\_completion\_list
  - ums\_lib.c, 35
  - ums\_lib.h, 44
- ums\_create\_scheduler
  - ums\_lib.c, 35
- ums\_create\_worker\_thread
  - ums\_lib.c, 36
  - ums\_lib.h, 44
- ums\_dequeue\_completion\_list\_items
  - ums\_lib.c, 36
  - ums\_lib.h, 45
- ums\_enter
  - ums\_lib.c, 36
  - ums\_lib.h, 45
- ums\_enter\_scheduling\_mode
  - ums\_lib.c, 37
  - ums\_lib.h, 45
- ums\_execute\_thread
  - ums\_lib.c, 37
  - ums\_lib.h, 46
- ums\_exit
  - ums\_lib.c, 37
  - ums\_lib.h, 46
- ums\_exit\_scheduling\_mode
  - ums\_lib.c, 38
  - ums\_lib.h, 46
- ums\_get\_next\_worker\_thread
  - ums\_lib.c, 38
  - ums\_lib.h, 46
- ums\_lib.c, 30
  - check\_if\_completion\_list\_exists, 33
  - check\_if\_scheduler\_exists, 34
  - check\_if\_worker\_exists, 34
  - cleanup, 34
  - close\_device, 34
  - completion\_lists, 39
  - open\_device, 35
  - schedulers, 40
  - ums\_create\_completion\_list, 35
  - ums\_create\_scheduler, 35
  - ums\_create\_worker\_thread, 36
  - ums\_dequeue\_completion\_list\_items, 36
  - ums\_enter, 36
  - ums\_enter\_scheduling\_mode, 37
  - ums\_execute\_thread, 37
  - ums\_exit, 37
  - ums\_exit\_scheduling\_mode, 38
  - ums\_get\_next\_worker\_thread, 38
  - ums\_thread\_exit, 38



- ums\_thread\_pause, [39](#)
- ums\_thread\_yield, [39](#)
- workers, [40](#)
- ums\_lib.h, [40](#)
  - check\_if\_completion\_list\_exists, [43](#)
  - check\_if\_scheduler\_exists, [43](#)
  - check\_if\_worker\_exists, [43](#)
  - cleanup, [44](#)
  - close\_device, [44](#)
  - open\_device, [44](#)
  - ums\_create\_completion\_list, [44](#)
  - ums\_create\_worker\_thread, [44](#)
  - ums\_dequeue\_completion\_list\_items, [45](#)
  - ums\_enter, [45](#)
  - ums\_enter\_scheduling\_mode, [45](#)
  - ums\_execute\_thread, [46](#)
  - ums\_exit, [46](#)
  - ums\_exit\_scheduling\_mode, [46](#)
  - ums\_get\_next\_worker\_thread, [46](#)
  - ums\_thread\_exit, [47](#)
  - ums\_thread\_pause, [47](#)
- ums\_scheduler, [10](#)
  - list\_params, [10](#)
  - sched\_params, [10](#)
  - tid, [11](#)
  - wid, [11](#)
- ums\_scheduler\_list, [11](#)
  - count, [11](#)
- ums\_thread\_exit
  - ums\_lib.c, [38](#)
  - ums\_lib.h, [47](#)
- ums\_thread\_pause
  - ums\_lib.c, [39](#)
  - ums\_lib.h, [47](#)
- ums\_thread\_yield
  - ums\_lib.c, [39](#)
- ums\_worker, [12](#)
  - state, [12](#)
  - wid, [12](#)
  - worker\_params, [12](#)
- ums\_worker\_list, [13](#)
  - count, [13](#)
- wid
  - ums\_scheduler, [11](#)
  - ums\_worker, [12](#)
- worker\_count
  - list\_params, [6](#)
  - ums\_completion\_list\_node, [10](#)
- worker\_params, [13](#)
  - clid, [14](#)
  - entry\_point, [14](#)
  - function\_args, [14](#)
  - stack\_addr, [14](#)
  - stack\_size, [14](#)
  - ums\_worker, [12](#)
- worker\_status
  - const.h, [18](#)
- workers
  - list\_params, [7](#)
  - ums\_lib.c, [40](#)