UMS: Library

Generated by Doxygen 1.9.2

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 2

File Index

2.1 File List

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

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

  Succesful 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.

Author

Bektur Umarbaev `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: ums_thread_pause() == ums_thread_yield(PAUSE)
FINISH	Used for completing a worker thread: ums_thread_exit() == ums_thread_yield(FINISH)

4.2 const.h

Go to the documentation of this file.

```

#pragma once

#include <linux/ioctl.h>

/*
 * Definitions
 */

#define UMS_NAME            "ums"
#define UMS_DEVICE          "/dev/ums"
#define UMS_IOC_MAGIC       'R'

/*
 * IOCTL definitions
 */
#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)

/*
 * Errors and return values
 */
#define UMS_SUCCESS                                             0

#define UMS_ERROR                                               1

#define UMS_ERROR_PROCESS_NOT_FOUND                             1000

#define UMS_ERROR_PROCESS_ALREADY_EXISTS                        1001
```

```
#define UMS_ERROR_COMPLETION_LIST_NOT_FOUND                             1002

#define UMS_ERROR_SCHEDULER_NOT_FOUND                                   1003

#define UMS_ERROR_WORKER_NOT_FOUND                                      1004

#define UMS_ERROR_STATE_RUNNING                                         1005

#define UMS_ERROR_CMD_IS_NOT_ISSUED_BY_MAIN_THREAD                      1006

#define UMS_ERROR_WORKER_ALREADY_RUNNING                                1007

#define UMS_ERROR_WRONG_INPUT                                           1008

#define UMS_ERROR_CMD_IS_NOT_ISSUED_BY_SCHEDULER                        1009

#define UMS_ERROR_CMD_IS_NOT_ISSUED_BY_WORKER                           1010

#define UMS_ERROR_WORKER_ALREADY_FINISHED                               1011

#define UMS_ERROR_NO_AVAILABLE_WORKERS                                  1012

#define UMS_ERROR_COMPLETION_LIST_ALREADY_FINISHED                      1013

#define UMS_ERROR_FAILED_TO_CREATE_PROC_ENTRY                           1014

#define UMS_ERROR_FAILED_TO_PROC_OPEN                                   1015

#define UMS_ERROR_COMPLETION_LIST_IS_USED_AND_CANNOT_BE_MODIFIED        1016


#define UMS_MIN_STACK_SIZE                                      4096

typedef enum state {
    IDLE,
    RUNNING,
    FINISHED
} state_t;

typedef enum worker_status {
    PAUSE,
    FINISH
} worker_status_t;

typedef unsigned int ums_sid_t;

typedef unsigned int ums_wid_t;

typedef unsigned int ums_clid_t;

typedef struct list_params {
    unsigned int size;
    unsigned int worker_count;
    state_t state;
    ums_wid_t workers[];
} list_params_t;

typedef struct worker_params {
    unsigned long entry_point;
    unsigned long function_args;
    unsigned long stack_size;
    unsigned long stack_addr;
    ums_clid_t clid;
} worker_params_t;

typedef struct scheduler_params {
    unsigned long entry_point;
    ums_clid_t clid;
    ums_sid_t sid;
    int core_id;
} 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.

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

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.

Author

Bektur Umarbaev 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);    \
(type *)( (char *)__mptr - offsetof(type,member) );})
```

Casts a member of a structure out to the containing structure

Parameters

ptr	the pointer to the member.
type	the type of the container struct this is embedded in.
member	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.

```

#ifndef _LINUX_LIST_H
#define _LINUX_LIST_H

#include <stdio.h>

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

#define container_of(ptr, type, member) ({                      \
        const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
        (type *)( (char *)__mptr - offsetof(type,member) );})
/*
 * These are non-NULL pointers that will result in page faults
 * under normal circumstances, used to verify that nobody uses
 * non-initialized list entries.
 */
#define LIST_POISON1  ((void *) 0x00100100)
#define LIST_POISON2  ((void *) 0x00200200)

struct list_head {
    struct list_head *next, *prev;
};

#define LIST_HEAD_INIT(name) { &(name), &(name) }

#define LIST_HEAD(name) \
    struct list_head name = LIST_HEAD_INIT(name)

#define INIT_LIST_HEAD(ptr) do { \
    (ptr)->next = (ptr); (ptr)->prev = (ptr); \
} while (0)

/*
 * Insert a new entry between two known consecutive entries.
 *
 * This is only for internal list manipulation where we know
 * the prev/next entries already!
 */
static inline void __list_add(struct list_head *new,
                  struct list_head *prev,
                  struct list_head *next)
{
    next->prev = new;
    new->next = next;
    new->prev = prev;
    prev->next = new;
}

static inline void list_add(struct list_head *new, struct list_head *head)
{
    __list_add(new, head, head->next);
}

static inline void list_add_tail(struct list_head *new, struct list_head *head)
{
    __list_add(new, head->prev, head);
}


/*
 * Delete a list entry by making the prev/next entries
 * point to each other.
 *
 * This is only for internal list manipulation where we know
 * the prev/next entries already!
 */
static inline void __list_del(struct list_head * prev, struct list_head * next)
{
    next->prev = prev;
    prev->next = next;
}

static inline void list_del(struct list_head *entry)
{
    __list_del(entry->prev, entry->next);
    entry->next = LIST_POISON1;
    entry->prev = LIST_POISON2;
}



static inline void list_del_init(struct list_head *entry)
```

```
{
    __list_del(entry->prev, entry->next);
    INIT_LIST_HEAD(entry);
}

static inline void list_move(struct list_head *list, struct list_head *head)
{
        __list_del(list->prev, list->next);
        list_add(list, head);
}

static inline void list_move_tail(struct list_head *list,
                  struct list_head *head)
{
        __list_del(list->prev, list->next);
        list_add_tail(list, head);
}

static inline int list_empty(const struct list_head *head)
{
    return head->next == head;
}

static inline void __list_splice(struct list_head *list,
                 struct list_head *head)
{
    struct list_head *first = list->next;
    struct list_head *last = list->prev;
    struct list_head *at = head->next;

    first->prev = head;
    head->next = first;

    last->next = at;
    at->prev = last;
}

static inline void list_splice(struct list_head *list, struct list_head *head)
{
    if (!list_empty(list))
        __list_splice(list, head);
}

static inline void list_splice_init(struct list_head *list,
                    struct list_head *head)
{
    if (!list_empty(list)) {
        __list_splice(list, head);
        INIT_LIST_HEAD(list);
    }
}

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

#define list_for_each(pos, head) \
  for (pos = (head)->next; pos != (head);   \
       pos = pos->next)

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

#define list_for_each_prev(pos, head) \
    for (pos = (head)->prev; prefetch(pos->prev), pos != (head); \
            pos = pos->prev)

#define list_for_each_safe(pos, n, head) \
    for (pos = (head)->next, n = pos->next; pos != (head); \
        pos = n, n = pos->next)

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

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

#define list_prepare_entry(pos, head, member) \
    ((pos) ? : list_entry(head, typeof(*pos), member))

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

```

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

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

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




/*
 * Double linked lists with a single pointer list head.
 * Mostly useful for hash tables where the two pointer list head is
 * too wasteful.
 * You lose the ability to access the tail in O(1).
 */

struct hlist_head {
    struct hlist_node *first;
};

struct hlist_node {
    struct hlist_node *next, **pprev;
};

#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)

static inline int hlist_unhashed(const struct hlist_node *h)
{
    return !h->pprev;
}

static inline int hlist_empty(const struct hlist_head *h)
{
    return !h->first;
}

static inline void __hlist_del(struct hlist_node *n)
{
    struct hlist_node *next = n->next;
    struct hlist_node **pprev = n->pprev;
    *pprev = next;
    if (next)
        next->pprev = pprev;
}

static inline void hlist_del(struct hlist_node *n)
{
    __hlist_del(n);
    n->next = LIST_POISON1;
    n->pprev = LIST_POISON2;
}


static inline void hlist_del_init(struct hlist_node *n)
{
    if (n->pprev)  {
        __hlist_del(n);
        INIT_HLIST_NODE(n);
    }
}

static inline void hlist_add_head(struct hlist_node *n, struct hlist_head *h)
{
    struct hlist_node *first = h->first;
    n->next = first;
    if (first)
        first->pprev = &n->next;
    h->first = n;
    n->pprev = &h->first;
}

```

```


/* next must be != NULL */
static inline void hlist_add_before(struct hlist_node *n,
                    struct hlist_node *next)
{
    n->pprev = next->pprev;
    n->next = next;
    next->pprev = &n->next;
    *(n->pprev) = n;
}

static inline void hlist_add_after(struct hlist_node *n,
                    struct hlist_node *next)
{
    next->next = n->next;
    n->next = next;
    next->pprev = &n->next;

    if(next->next)
        next->next->pprev  = &next->next;
}



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

#define hlist_for_each(pos, head) \
    for (pos = (head)->first; pos && ({ prefetch(pos->next); 1; }); \
         pos = pos->next)

#define hlist_for_each_safe(pos, n, head) \
    for (pos = (head)->first; pos && ({ n = pos->next; 1; }); \
         pos = n)

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

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

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

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


#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 comletion 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 pthreds 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.

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

Author

Bektur Umarbaev 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

clid	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

wid	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 pthreds 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

clid	ID of the completion list that is assigned to the scheduler
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

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 comletion list Library requests UMS kernel module to create a worker thread by passing worker_params.

Parameters

clid	ID of the completion list where worker thread is assigned to
stack_size	Stack size of the worker thread set by a user
entry_point	Function pointer and an entry point set by a user, that serves as a starting point of the worker thread
args	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

args	Pointer to scheduler_params 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

wid	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

list	List parameter that is created after ums_dequeue_completion_list_items() 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

status	defines the status of the execution flow of the worker thread (passing `PAUSE` will pause the execution, when `FINISH` 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 comletion 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.

Author

Bektur Umarbaev `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

clid	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

wid	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

clid	ID of the completion list where worker thread is assigned to
stack_size	Stack size of the worker thread set by a user
entry_point	Function pointer and an entry point set by a user, that serves as a starting point of the worker thread
args	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 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 succesful 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 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

args	Pointer to scheduler_params 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

wid	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

list	List parameter that is created after ums_dequeue_completion_list_items() 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.

```c

#pragma once

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

#define UMS_DEVICE "/dev/ums"

typedef struct ums_completion_list ums_completion_list_t;
typedef struct ums_completion_list_node ums_completion_list_node_t;
typedef struct ums_worker ums_worker_t;
typedef struct ums_worker_list ums_worker_list_t;
typedef struct ums_scheduler_list ums_scheduler_list_t;
typedef struct ums_scheduler ums_scheduler_t;

int ums_enter();
int ums_exit();

ums_clid_t ums_create_completion_list();
ums_wid_t ums_create_worker_thread(ums_clid_t clid, unsigned long stack_size, void (*entry_point)(void *), void *args);
ums_sid_t ums_create_scheduler(ums_clid_t clid, void (*entry_point)(void *));
void *ums_enter_scheduling_mode(void *args);
int ums_exit_scheduling_mode();
int ums_execute_thread(ums_wid_t wid);
int ums_thread_yield();
int ums_thread_pause();
int ums_thread_exit();
list_params_t *ums_dequeue_completion_list_items();
ums_wid_t ums_get_next_worker_thread(list_params_t *list);

int open_device();
int close_device();
int cleanup();
ums_completion_list_node_t *check_if_completion_list_exists(ums_clid_t clid);
ums_worker_t *check_if_worker_exists(ums_wid_t wid);
ums_scheduler_t *check_if_scheduler_exists();

typedef struct ums_completion_list {
    struct list_head list;
    unsigned int count;
} ums_completion_list_t;

typedef struct ums_completion_list_node {
    ums_clid_t clid;
    state_t state;
    unsigned int worker_count;
    struct list_head list;
    list_params_t *list_params;
} ums_completion_list_node_t;

typedef struct ums_worker_list {
    struct list_head list;
    unsigned int count;
} ums_worker_list_t;

typedef struct ums_worker {
    ums_wid_t wid;
    state_t state;
    struct list_head list;
    worker_params_t *worker_params;
} ums_worker_t;

typedef struct ums_scheduler_list {
    struct list_head list;
    unsigned int count;
} ums_scheduler_list_t;

typedef struct ums_scheduler {
    struct list_head list;
    pthread_t tid;
    ums_wid_t wid;
    scheduler_params_t *sched_params;
    list_params_t *list_params;
} ums_scheduler_t;

#define init(type) (type*)malloc(sizeof(type))
#define delete(val) free(val)
#define create_list_params(size) (list_params_t*)malloc(sizeof(list_params_t) + size * sizeof(ums_wid_t))
```

Index