User Mode thread Scheduling (user library)

Generated by Doxygen 1.9.3

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 buff_cq Struct Reference

contain the biffer used for the operation of the completion queue

```
#include <ums.h>
```

Data Fields

- int pids [COMPLETION_QUEUE_BUFF]

3.1.1 Detailed Description

contain the biffer used for the operation of the completion queue

3.1.2 Field Documentation

3.1.2.1 pids

```
int buff_cq::pids[COMPLETION_QUEUE_BUFF]
```

this array is the buffer

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

- ums.h

3.2 cq_list_item Struct Reference

Data Fields

- int id
- int used_by
- ums_cq_param_t cq_item
- struct list_head list

3.2.1 Field Documentation

3.2.1.1 cq_item

```
ums_cq_param_t cq_list_item::cq_item
```

param used in the ioctl call

3.2.1.2 id

```
int cq_list_item::id
```

id of the completion queue

3.2.1.3 list

```
struct list_head cq_list_item::list
```

list head for the list api

3.2.1.4 used_by

```
int cq_list_item::used_by
```

number of the ums that use it

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

- ums.h

3.3 list_head Struct Reference

Data Fields

- struct list_head ∗ **next**
- struct list_head ∗ **prev**

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

- list.h

3.4 pthread_entry Struct Reference

Data Fields

- pthread_t tid
- struct list_head list

3.4.1 Field Documentation

3.4.1.1 list

```
struct list_head pthread_entry::list
```

list head for the list api

3.4.1.2 tid

```
pthread_t pthread_entry::tid
```

thread id of the ums used in the ExitFromUmsSchedulingMode function

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

- ums.h

3.5 ums_entry_info Struct Reference

Data Fields

- ums_entry_point entry
- int cq_id
- int owner_pid
- int ret_value

3.5.1 Field Documentation

3.5.1.1 cq_id

```
int ums_entry_info::cq_id
```

∗id of the completion queue used

3.5.1.2 entry

```
ums_entry_point ums_entry_info::entry
```

entry point function

3.5.1.3 owner_pid

```
int ums_entry_info::owner_pid
```

pid of the owner of the ums

3.5.1.4 ret_value

```
int ums_entry_info::ret_value
```

ret value of the initialization

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

- ums.h

3.6 ums_scheduler Struct Reference

Data Fields

- pthread_t ∗ **ums_threads_list**
- int **n_cpu**
- int **cq_id**

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

- ums.h

3.7 worker_thread_job_info Struct Reference

Data Fields

- worker_job job
- void ∗ args_routine
- int pid

3.7.1 Field Documentation

3.7.1.1 args_routine

```
void* worker_thread_job_info::args_routine
```

pointer to the args of the job

3.7.1.2 job

```
worker_job worker_thread_job_info::job
```

poiner to the job (finction) of the worker thread

3.7.1.3 pid

```
int worker_thread_job_info::pid
```

pid of the worker thread

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

- ums.h

Chapter 4

File Documentation

4.1 list.h File Reference

This file is the kernel implementation of the list is from Linux Kernel (include/linux/list.h)

```
#include <stddef.h>
```

Data Structures

- struct list_head

Macros

- #define **offsetof**(TYPE, MEMBER) ((size_t) &((TYPE ∗)0)->MEMBER)
- #define container_of(ptr, type, member)
- #define **LIST_HEAD_INIT**(name) { &(name), &(name) }
- #define **LIST_HEAD**(name) struct list_head name = LIST_HEAD_INIT(name)
- #define list_entry(ptr, type, member) container_of(ptr, type, member)
- #define list_first_entry(ptr, type, member) list_entry((ptr)->next, type, member)
- #define list_last_entry(ptr, type, member) list_entry((ptr)->prev, type, member)
- #define list_next_entry(pos, member) list_entry((pos)->member.next, typeof(∗(pos)), member)
- #define list_prev_entry(pos, member) list_entry((pos)->member.prev, typeof(∗(pos)), member)
- #define list_for_each(pos, head) for (pos = (head)->next; pos != (head); pos = pos->next)
- #define list_for_each_continue(pos, head) for (pos = pos->next; pos != (head); pos = pos->next)
- #define list_for_each_prev(pos, head) for (pos = (head)->prev; pos != (head); pos = pos->prev)
- #define list_for_each_safe(pos, n, head)
- #define list_for_each_prev_safe(pos, n, head)
- #define list_entry_is_head(pos, head, member) (&pos->member == (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_continue_reverse(pos, head, member)
- #define list_for_each_entry_from(pos, head, member)
- #define list_for_each_entry_from_reverse(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_from(pos, n, head, member)
- #define list_for_each_entry_safe_reverse(pos, n, head, member)
- #define list_safe_reset_next(pos, n, member) n = list_next_entry(pos, member)

4.1.1 Detailed Description

This file is the kernel implementation of the list is from Linux Kernel (include/linux/list.h)

4.1.2 Macro Definition Documentation

4.1.2.1 container_of

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

Value:

```
({                \
(type *)((char *)ptr - offsetof(type, member)); })
```

container_of - cast 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.

4.1.2.2 list_entry

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

list_entry - get the struct for this entry

Parameters

ptr	the &struct list_head pointer.
type	the type of the struct this is embedded in.
member	the name of the list_head within the struct.

4.1.2.3 list_entry_is_head

```
#define list_entry_is_head(
              pos,
              head,
              member )   (&pos->member == (head))
```

list_entry_is_head - test if the entry points to the head of the list

Parameters

pos	the type ∗ to cursor
head	the head for your list.
member	the name of the list_head within the struct.

4.1.2.4 list_first_entry

```
#define list_first_entry(
              ptr,
              type,
              member )   list_entry((ptr)->next, type, member)
```

list_first_entry - get the first element from a list

Parameters

ptr	the list head to take the element from.
type	the type of the struct this is embedded in.
member	the name of the list_head within the struct.

Note, that list is expected to be not empty.

4.1.2.5 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

Parameters

pos	the &struct list_head to use as a loop cursor.
head	the head for your list.

4.1.2.6 list_for_each_continue

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

list_for_each_continue - continue iteration over a list

Parameters

pos	the &struct list_head to use as a loop cursor.
head	the head for your list.

Continue to iterate over a list, continuing after the current position.

4.1.2.7 list_for_each_entry

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

Value:

```
    for (pos = list_first_entry(head, typeof(*pos), member);    \
         !list_entry_is_head(pos, head, member);          \
         pos = list_next_entry(pos, member))
```

list_for_each_entry - iterate over list of given type

Parameters

pos	the type ∗ to use as a loop cursor.
head	the head for your list.
member	the name of the list_head within the struct.

4.1.2.8 list_for_each_entry_continue

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

Value:

```
    for (pos = list_next_entry(pos, member);           \
         !list_entry_is_head(pos, head, member);          \
         pos = list_next_entry(pos, member))
```

list_for_each_entry_continue - continue iteration over list of given type

Parameters

pos	the type ∗ to use as a loop cursor.
head	the head for your list.
member	the name of the list_head within the struct.

Continue to iterate over list of given type, continuing after the current position.

4.1.2.9 list_for_each_entry_continue_reverse

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

Value:

```
    for (pos = list_prev_entry(pos, member);          \
         !list_entry_is_head(pos, head, member);          \
         pos = list_prev_entry(pos, member))
```

list_for_each_entry_continue_reverse - iterate backwards from the given point

Parameters

pos	the type * to use as a loop cursor.
head	the head for your list.
member	the name of the list_head within the struct.

Start to iterate over list of given type backwards, continuing after the current position.

4.1.2.10 list_for_each_entry_from

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

Value:

```
    for (; !list_entry_is_head(pos, head, member);            \
         pos = list_next_entry(pos, member))
```

list_for_each_entry_from - iterate over list of given type from the current point

Parameters

pos	the type * to use as a loop cursor.
head	the head for your list.
member	the name of the list_head within the struct.

Iterate over list of given type, continuing from current position.

4.1.2.11 list_for_each_entry_from_reverse

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

Value:

```
    for (; !list_entry_is_head(pos, head, member);            \
         pos = list_prev_entry(pos, member))
```

list_for_each_entry_from_reverse - iterate backwards over list of given type from the current point

Parameters

pos	the type ∗ to use as a loop cursor.
head	the head for your list.
member	the name of the list_head within the struct.

Iterate backwards over list of given type, continuing from current position.

4.1.2.12 list_for_each_entry_reverse

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

Value:

```
    for (pos = list_last_entry(head, typeof(*pos), member);      \
         !list_entry_is_head(pos, head, member);           \
         pos = list_prev_entry(pos, member))
```

list_for_each_entry_reverse - iterate backwards over list of given type.

Parameters

pos	the type ∗ to use as a loop cursor.
head	the head for your list.
member	the name of the list_head within the struct.

4.1.2.13 list_for_each_entry_safe

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

Value:

```
    for (pos = list_first_entry(head, typeof(*pos), member),    \
        n = list_next_entry(pos, member);            \
         !list_entry_is_head(pos, head, member);           \
         pos = n, n = list_next_entry(n, member))
```

list_for_each_entry_safe - iterate over list of given type safe against removal of list entry

Parameters

pos	the type ∗ to use as a loop cursor.
n	another type ∗ to use as temporary storage
head	the head for your list.
member	the name of the list_head within the struct.

4.1.2.14 list_for_each_entry_safe_continue

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

Value:

```
    for (pos = list_next_entry(pos, member),                  \
        n = list_next_entry(pos, member);                 \
         !list_entry_is_head(pos, head, member);              \
         pos = n, n = list_next_entry(n, member))
```

list_for_each_entry_safe_continue - continue list iteration safe against removal

Parameters

pos	the type ∗ to use as a loop cursor.
n	another type ∗ to use as temporary storage
head	the head for your list.
member	the name of the list_head within the struct.

Iterate over list of given type, continuing after current point, safe against removal of list entry.

4.1.2.15 list_for_each_entry_safe_from

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

Value:

```
    for (n = list_next_entry(pos, member);                    \
         !list_entry_is_head(pos, head, member);              \
         pos = n, n = list_next_entry(n, member))
```

list_for_each_entry_safe_from - iterate over list from current point safe against removal

Parameters

pos	the type ∗ to use as a loop cursor.
n	another type ∗ to use as temporary storage
head	the head for your list.
member	the name of the list_head within the struct.

Iterate over list of given type from current point, safe against removal of list entry.

4.1.2.16 list_for_each_entry_safe_reverse

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

Value:

```
    for (pos = list_last_entry(head, typeof(*pos), member),      \
        n = list_prev_entry(pos, member);            \
         !list_entry_is_head(pos, head, member);          \
         pos = n, n = list_prev_entry(n, member))
```

list_for_each_entry_safe_reverse - iterate backwards over list safe against removal

Parameters

pos	the type ∗ to use as a loop cursor.
n	another type ∗ to use as temporary storage
head	the head for your list.
member	the name of the list_head within the struct.

Iterate backwards over list of given type, safe against removal of list entry.

4.1.2.17 list_for_each_prev

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

list_for_each_prev - iterate over a list backwards

Parameters

pos	the &struct list_head to use as a loop cursor.
head	the head for your list.

4.1.2.18 list_for_each_prev_safe

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

Value:

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

list_for_each_prev_safe - iterate over a list backwards safe against removal of list entry

Parameters

pos	the &struct list_head to use as a loop cursor.
n	another &struct list_head to use as temporary storage
head	the head for your list.

4.1.2.19 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

Parameters

pos	the &struct list_head to use as a loop cursor.
n	another &struct list_head to use as temporary storage
head	the head for your list.

4.1.2.20 list_last_entry

```
#define list_last_entry(
                ptr,
                type,
                member )   list_entry((ptr)->prev, type, member)
```

list_last_entry - get the last element from a list

Parameters

ptr	the list head to take the element from.
type	the type of the struct this is embedded in.
member	the name of the list_head within the struct.

Note, that list is expected to be not empty.

4.1.2.21 list_next_entry

```
#define list_next_entry(
              pos,
              member )   list_entry((pos)->member.next, typeof(*(pos)), member)
```

list_next_entry - get the next element in list

Parameters

pos	the type ∗ to cursor
member	the name of the list_head within the struct.

4.1.2.22 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 in list_for_each_entry_continue()

Parameters

pos	the type ∗ to use as a start point
head	the head of the list
member	the name of the list_head within the struct.

Prepares a pos entry for use as a start point in list_for_each_entry_continue().

4.1.2.23 list_prev_entry

```
#define list_prev_entry(
              pos,
              member )   list_entry((pos)->member.prev, typeof(*(pos)), member)
```

list_prev_entry - get the prev element in list

Parameters

pos	the type ∗ to cursor
member	the name of the list_head within the struct.

4.1.2.24 list_safe_reset_next

```
#define list_safe_reset_next(
              pos,
              n,
              member )   n = list_next_entry(pos, member)
```

list_safe_reset_next - reset a stale list_for_each_entry_safe loop

Parameters

pos	the loop cursor used in the list_for_each_entry_safe loop
n	temporary storage used in list_for_each_entry_safe
member	the name of the list_head within the struct.

list_safe_reset_next is not safe to use in general if the list may be modified concurrently (eg. the lock is dropped in the loop body). An exception to this is if the cursor element (pos) is pinned in the list, and list_safe_reset_next is called after re-taking the lock and before completing the current iteration of the loop body.

4.2 list.h

Go to the documentation of this file.

```

/* SPDX-License-Identifier: GPL-2.0 */
#ifndef _LINUX_LIST_H
#define _LINUX_LIST_H
#include <stddef.h>

/*
 * Circular 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.
 */

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

#ifndef container_of
#define container_of(ptr, type, member) ({          \
    (type *)((char *)ptr - offsetof(type, member)); })
#endif

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)

static inline void INIT_LIST_HEAD(struct list_head *list)
{
    do{
        list->next = list;
        list->prev = list;
    }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;
}

/*
 * Delete a list entry and clear the ’prev’ pointer.
 *
 * This is a special-purpose list clearing method used in the networking code
 * for lists allocated as per-cpu, where we don’t want to incur the extra
 * WRITE_ONCE() overhead of a regular list_del_init(). The code that uses this
 * needs to check the node ’prev’ pointer instead of calling list_empty().
 */
static inline void __list_del_clearprev(struct list_head *entry)
{
    __list_del(entry->prev, entry->next);
    entry->prev = NULL;
}

static inline void __list_del_entry(struct list_head *entry)
{

    __list_del(entry->prev, entry->next);
}

static inline void list_del(struct list_head *entry)
{
    __list_del_entry(entry);
    entry->next = (void *)0;
    entry->prev = (void *)0;
}

static inline void list_replace(struct list_head *old,
                struct list_head *new)
{
    new->next = old->next;
    new->next->prev = new;
    new->prev = old->prev;
    new->prev->next = new;
}

static inline void list_replace_init(struct list_head *old,
                     struct list_head *new)
{
    list_replace(old, new);
    INIT_LIST_HEAD(old);
}

static inline void list_swap(struct list_head *entry1,
                 struct list_head *entry2)
{
    struct list_head *pos = entry2->prev;

    list_del(entry2);
    list_replace(entry1, entry2);
    if (pos == entry1)
```

```
        pos = entry2;
    list_add(entry1, pos);
}

static inline void list_del_init(struct list_head *entry)
{
    __list_del_entry(entry);
    INIT_LIST_HEAD(entry);
}

static inline void list_move(struct list_head *list, struct list_head *head)
{
    __list_del_entry(list);
    list_add(list, head);
}

static inline void list_move_tail(struct list_head *list,
                  struct list_head *head)
{
    __list_del_entry(list);
    list_add_tail(list, head);
}

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

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

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

static inline int list_is_first(const struct list_head *list,
                    const struct list_head *head)
{
    return list->prev == head;
}

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

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

static inline void list_rotate_left(struct list_head *head)
{
    struct list_head *first;

    if (!list_empty(head)) {
        first = head->next;
        list_move_tail(first, head);
    }
}

static inline void list_rotate_to_front(struct list_head *list,
                    struct list_head *head)
{
    /*
     * Deletes the list head from the list denoted by @head and
     * places it as the tail of @list, this effectively rotates the
     * list so that @list is at the front.
     */
    list_move_tail(head, list);
}

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

static inline void __list_cut_position(struct list_head *list,
        struct list_head *head, struct list_head *entry)
{
    struct list_head *new_first = entry->next;
    list->next = head->next;
    list->next->prev = list;
    list->prev = entry;
    entry->next = list;
    head->next = new_first;
    new_first->prev = head;
}

static inline void list_cut_position(struct list_head *list,
        struct list_head *head, struct list_head *entry)
{
    if (list_empty(head))
        return;
    if (list_is_singular(head) &&
        (head->next != entry && head != entry))
        return;
    if (entry == head)
        INIT_LIST_HEAD(list);
    else
        __list_cut_position(list, head, entry);
}

static inline void list_cut_before(struct list_head *list,
                   struct list_head *head,
                   struct list_head *entry)
{
    if (head->next == entry) {
        INIT_LIST_HEAD(list);
        return;
    }
    list->next = head->next;
    list->next->prev = list;
    list->prev = entry->prev;
    list->prev->next = list;
    head->next = entry;
    entry->prev = head;
}

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

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

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

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

static inline void list_splice_tail(struct list_head *list,
                struct list_head *head)
{
    if (!list_empty(list))
        __list_splice(list, head->prev, head);
}

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

static inline void list_splice_tail_init(struct list_head *list,
                     struct list_head *head)
{
    if (!list_empty(list)) {
        __list_splice(list, head->prev, head);
        INIT_LIST_HEAD(list);
    }
}

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

#define list_first_entry(ptr, type, member) \
    list_entry((ptr)->next, type, member)

```

```
#define list_last_entry(ptr, type, member) \
    list_entry((ptr)->prev, type, member)

#define list_next_entry(pos, member) \
    list_entry((pos)->member.next, typeof(*(pos)), member)

#define list_prev_entry(pos, member) \
    list_entry((pos)->member.prev, typeof(*(pos)), member)

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

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

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

#define list_entry_is_head(pos, head, member)               \
    (&pos->member == (head))

#define list_for_each_entry(pos, head, member)              \
    for (pos = list_first_entry(head, typeof(*pos), member);    \
         !list_entry_is_head(pos, head, member);            \
         pos = list_next_entry(pos, member))

#define list_for_each_entry_reverse(pos, head, member)          \
    for (pos = list_last_entry(head, typeof(*pos), member);     \
         !list_entry_is_head(pos, head, member);            \
         pos = list_prev_entry(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_next_entry(pos, member);            \
         !list_entry_is_head(pos, head, member);            \
         pos = list_next_entry(pos, member))

#define list_for_each_entry_continue_reverse(pos, head, member)     \
    for (pos = list_prev_entry(pos, member);            \
         !list_entry_is_head(pos, head, member);            \
         pos = list_prev_entry(pos, member))

#define list_for_each_entry_from(pos, head, member)             \
    for (; !list_entry_is_head(pos, head, member);          \
         pos = list_next_entry(pos, member))

#define list_for_each_entry_from_reverse(pos, head, member)     \
    for (; !list_entry_is_head(pos, head, member);          \
         pos = list_prev_entry(pos, member))

#define list_for_each_entry_safe(pos, n, head, member)          \
    for (pos = list_first_entry(head, typeof(*pos), member),    \
        n = list_next_entry(pos, member);           \
         !list_entry_is_head(pos, head, member);            \
         pos = n, n = list_next_entry(n, member))

#define list_for_each_entry_safe_continue(pos, n, head, member)         \
    for (pos = list_next_entry(pos, member),                \
        n = list_next_entry(pos, member);               \
         !list_entry_is_head(pos, head, member);                \
         pos = n, n = list_next_entry(n, member))

#define list_for_each_entry_safe_from(pos, n, head, member)             \
    for (n = list_next_entry(pos, member);                  \
         !list_entry_is_head(pos, head, member);                \
         pos = n, n = list_next_entry(n, member))

#define list_for_each_entry_safe_reverse(pos, n, head, member)      \
    for (pos = list_last_entry(head, typeof(*pos), member),     \
        n = list_prev_entry(pos, member);           \
         !list_entry_is_head(pos, head, member);            \
         pos = n, n = list_prev_entry(n, member))

#define list_safe_reset_next(pos, n, member)                \
    n = list_next_entry(pos, member)

```

```

#endif
```

4.3 ums.c File Reference

This file contains main definiton and function for the ums user library.

```
#include "ums.h"
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <assert.h>
```

Functions

- **LIST_HEAD** (global_cq_list)
- void ∗ __ums_entry_point_wrapper (void ∗args)

  wrapper function of the ums it initialize the ums before call the entry point
- void ∗ __default_entry_point (void ∗arguments)

  the default entry poont for the ums schedulers
- int CreateNewWorker (worker_job job_to_perform, void ∗job_args)

  Create a New Worker thread. It busy wait until the pid entry in the new_job_struct is populated or is elapsed delta time. It return the pid of the new worker thread or -1 in case of error.
- int UmsThreadYield ()

  called from a worker thread, it pauses the execution of the current thread and the UMS scheduler entry point is executed for determining the next thread to be scheduled
- int ExecuteUmsThread (unsigned worker_id)

  called from a scheduler thread, it executes the passed worker thread by switching the entire context
- pthread_t UMS_thread_create (ums_entry_point entry_point, int completion_queue_id, int n_cpu)

  converts a standard pthread in a UMS Scheduler thread, the function takes as input a completion list of worker threads and a entry point function
- ums_t ∗ EnterUmsSchedulingMode (ums_entry_point entry_point, int completion_queue_id)

  create N ums scheduler thread (N nuber of cores in the computer) and will scheduke the thred from the completion queue id
- void ExitFromUmsSchedulingMode (ums_t ∗ums)

  Exit from UMS mode.
- int CreateCompletionQueue ()

  Create a Completion Queue object and return the completion queue id. During this process it also init the data structure to buffer the worker thread.
- int AppendToCompletionQueue (int completion_queue_id, int worker_pid)

  it insert a worker pid inside a completion queue berfore it perform some check in order to see if the completion queue exist
- int FlushCompletionQueue (int completion_queue_id)

  actually insert the worker pid into the data structure in the kernel using the device ioctl
- int DequeueUmsCompletionListItems (dequeued_cq_t ∗return_cq)

  dequeue the first 100 pid of the workers inside the return queue
- void **resetUMSFlag** (void)
- **__attribute__** ((constructor))

  initialize the dev semaphore
- **__attribute__** ((destructor))

  close the device file

Variables

- sem_t **ums_dev_sem**
- int **global_ums_fd** = -1
- volatile bool **ums_mode_enabled** = FALSE

4.3.1 Detailed Description

This file contains main definiton and function for the ums user library.

Author

Tiziano Colagrossi tiziano.colagrossi@gmail.com

4.3.2 Function Documentation

4.3.2.1 __default_entry_point()

```
void * __default_entry_point (
            void * arguments )
```

the default entry poont for the ums schedulers

Parameters

arguments	

Returns

void*

4.3.2.2 __ums_entry_point_wrapper()

```
void * __ums_entry_point_wrapper (
            void * args )
```

wrapper function of the ums it initialize the ums before call the entry point

Parameters

args	pointer to ums_entry_info_t struct

Returns

void∗

4.3.2.3 AppendToCompletionQueue()

```
int AppendToCompletionQueue (
            int completion_queue_id,
            int worker_pid )
```

it insert a worker pid inside a completion queue berfore it perform some check in order to see if the completion queue exist

Parameters

completion_queue← _id	id of the completion queue where apped the worker
worker_pid	pid of the worker

Returns

int

4.3.2.4 CreateCompletionQueue()

```
int CreateCompletionQueue (
            void )
```

Create a Completion Queue object and return the completion queue id. During this process it also init the data structure to buffer the worker thread.

Returns

int

4.3.2.5 CreateNewWorker()

```
int CreateNewWorker (
            worker_job job_to_perform,
            void * job_args )
```

Create a New Worker thread. It busy wait until the pid entry in the new_job_struct is populated or is elapsed delta time. It return the pid of the new worker thread or -1 in case of error.

Parameters

job_to_perform	job function of the worker thread
job_args	args used from the job function (optional)

Returns

int

4.3.2.6 DequeueUmsCompletionListItems()

```
int DequeueUmsCompletionListItems (
            dequeued_cq_t * return_cq )
```

dequeue the first 100 pid of the workers inside the return queue

Parameters

return_cq	pointer to dequeued_cq_t struct

Returns

int

4.3.2.7 EnterUmsSchedulingMode()

```
ums_t * EnterUmsSchedulingMode (
            ums_entry_point entry_point,
            int completion_queue_id )
```

create N ums scheduler thread (N nuber of cores in the computer) and will scheduke the thred from the completion queue id

Parameters

entry_point	entry_point for the sceduler uf null will use the default
completion_queue←_id	id of the completion queue used by the ums

Returns

ums_t∗

4.3.2.8 ExecuteUmsThread()

```
int ExecuteUmsThread (
            unsigned worker_id )
```

called from a scheduler thread, it executes the passed worker thread by switching the entire context

Parameters

worker_id	pid of the worker that will be executed

Returns

int

4.3.2.9 ExitFromUmsSchedulingMode()

```
void ExitFromUmsSchedulingMode (
            ums_t * ums )
```

Exit from UMS mode.

wait for all the ums to end and then free the data structured used

Parameters

ums	

4.3.2.10 FlushCompletionQueue()

```
int FlushCompletionQueue (
            int completion_queue_id )
```

actually insert the worker pid into the data structure in the kernel using the device ioctl

Parameters

completion_queue_id	id of the completion queue

Returns

int

4.3.2.11 UMS_thread_create()

```
pthread_t UMS_thread_create (
            ums_entry_point entry_point,
            int completion_queue_id,
            int n_cpu )
```

converts a standard pthread in a UMS Scheduler thread, the function takes as input a completion list of worker threads and a entry point function

Parameters

entry_point	entry point funtion of the ums
completion_queue↩ _id	id of the completion queue used by the ums
n_cpu	cpu where this ums will be scheduled

Returns

int

4.3.2.12 UmsThreadYield()

```
int UmsThreadYield (
            void )
```

called from a worker thread, it pauses the execution of the current thread and the UMS scheduler entry point is executed for determining the next thread to be scheduled

Returns

int

4.4 ums.h File Reference

This file is the header of the user library.

```
#include "../kernel_module/shared.h"
#include "list.h"
#include <stdlib.h>
#include <semaphore.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/syscall.h>
#include <time.h>
```

Data Structures

- struct buff_cq

  contain the biffer used for the operation of the completion queue
- struct cq_list_item
- struct worker_thread_job_info
- struct ums_entry_info
- struct pthread_entry
- struct ums_scheduler

Macros

- #define **_GNU_SOURCE**
- #define **TRUE** 1
- #define **FALSE** 0
- #define **GENERAL_UMS_ERROR** -1
- #define **EXIT_UMS_MOD** -2
- #define **UMS_PATH** "/dev/ums"
- #define **MODULE_UMSLIB_LOG** "[UMS LIB DEBUG]: "
- #define **UMSLIB_DEBUG**
- #define **__F_APPEND** "__append_new_worker_to_cq: "
- #define **F_APPEND** "AppendToCompletionQueue: "
- #define **F_FLUSH** "FlushCompletionQueue: "
- #define INIT_UMS_ENTRY_STRUCT(X, E, I, O)

Typedefs

- typedef void ∗(∗ **worker_job**) (void ∗)
- typedef void ∗(∗ **ums_entry_point**) (void ∗)
- typedef int **bool**
- typedef struct buff_cq **dequeued_cq_t**

  contain the biffer used for the operation of the completion queue
- typedef struct cq_list_item **cq_list_item_t**
- typedef struct worker_thread_job_info **worker_thread_job_info_t**
- typedef struct ums_entry_info **ums_entry_info_t**
- typedef struct pthread_entry **pthread_entry_t**
- typedef struct ums_scheduler **ums_t**

Functions

- int CreateNewWorker (worker_job job_to_perform, void ∗job_args)

  Create a New Worker thread. It busy wait until the pid entry in the new_job_struct is populated or is elapsed delta time. It return the pid of the new worker thread or -1 in case of error.
- int UmsThreadYield (void)

  called from a worker thread, it pauses the execution of the current thread and the UMS scheduler entry point is executed for determining the next thread to be scheduled
- int ExecuteUmsThread (unsigned worker_id)

  called from a scheduler thread, it executes the passed worker thread by switching the entire context
- pthread_t UMS_thread_create (ums_entry_point entry_point, int completion_queue_id, int n_cpu)

  converts a standard pthread in a UMS Scheduler thread, the function takes as input a completion list of worker threads and a entry point function

- ums_t ∗ EnterUmsSchedulingMode (ums_entry_point entry_point, int completion_queue_id)

  create N ums scheduler thread (N nuber of cores in the computer) and will scheduke the thred from the completion queue id
- void ExitFromUmsSchedulingMode (ums_t ∗ums)

  Exit from UMS mode.
- int CreateCompletionQueue (void)

  Create a Completion Queue object and return the completion queue id. During this process it also init the data structure to buffer the worker thread.
- int AppendToCompletionQueue (int completion_queue_id, int worker_pid)

  it insert a worker pid inside a completion queue berfore it perform some check in order to see if the completion queue exist
- int FlushCompletionQueue (int completion_queue_id)

  actually insert the worker pid into the data structure in the kernel using the device ioctl
- int DequeueUmsCompletionListItems (dequeued_cq_t ∗return_cq)

  dequeue the first 100 pid of the workers inside the return queue
- void **resetUMSFlag** (void)

4.4.1 Detailed Description

This file is the header of the user library.

Author

Tiziano Colagrossi tiziano.colagrossi@gmail.com

4.4.2 Macro Definition Documentation

4.4.2.1 INIT_UMS_ENTRY_STRUCT

```
#define INIT_UMS_ENTRY_STRUCT(
                X,
                E,
                I,
                O )
```

Value:

```
ums_entry_info_t X = {       \
        .entry = E,     \
        .cq_id = I,      \
        .ret_value = 1, \
        .owner_pid = 0  \
        }
```

4.4.3 Function Documentation

4.4.3.1 AppendToCompletionQueue()

```
int AppendToCompletionQueue (
              int completion_queue_id,
              int worker_pid )
```

it insert a worker pid inside a completion queue berfore it perform some check in order to see if the completion queue exist

Parameters

completion_queue↩ _id	id of the completion queue where apped the worker
worker_pid	pid of the worker

Returns

int

4.4.3.2 CreateCompletionQueue()

```
int CreateCompletionQueue (
            void  )
```

Create a Completion Queue object and return the completion queue id. During this process it also init the data structure to buffer the worker thread.

Returns

int

4.4.3.3 CreateNewWorker()

```
int CreateNewWorker (
            worker_job job_to_perform,
            void * job_args )
```

Create a New Worker thread. It busy wait until the pid entry in the new_job_struct is populated or is elapsed delta time. It return the pid of the new worker thread or -1 in case of error.

Parameters

job_to_perform	job function of the worker thread
job_args	args used from the job function (optional)

Returns

int

4.4.3.4 DequeueUmsCompletionListItems()

```
int DequeueUmsCompletionListItems (
            dequeued_cq_t * return_cq )
```

dequeue the first 100 pid of the workers inside the return queue

Parameters

return_cq	pointer to dequeued_cq_t struct

Returns

int

4.4.3.5 EnterUmsSchedulingMode()

```
ums_t * EnterUmsSchedulingMode (
            ums_entry_point entry_point,
            int completion_queue_id )
```

create N ums scheduler thread (N nuber of cores in the computer) and will scheduke the thred from the completion queue id

Parameters

entry_point	entry_point for the sceduler uf null will use the default
completion_queue↩ _id	id of the completion queue used by the ums

Returns

ums_t∗

4.4.3.6 ExecuteUmsThread()

```
int ExecuteUmsThread (
            unsigned worker_id )
```

called from a scheduler thread, it executes the passed worker thread by switching the entire context

Parameters

worker↩ _id	pid of the worker that will be executed

Returns

int

4.4.3.7 ExitFromUmsSchedulingMode()

```
void ExitFromUmsSchedulingMode (
            ums_t * ums )
```

Exit from UMS mode.

wait for all the ums to end and then free the data structured used

Parameters

ums	

4.4.3.8 FlushCompletionQueue()

```
int FlushCompletionQueue (
            int completion_queue_id )
```

actually insert the worker pid into the data structure in the kernel using the device ioctl

Parameters

completion_queue↩_id	id of the completion queue

Returns

int

4.4.3.9 UMS_thread_create()

```
pthread_t UMS_thread_create (
            ums_entry_point entry_point,
            int completion_queue_id,
            int n_cpu )
```

converts a standard pthread in a UMS Scheduler thread, the function takes as input a completion list of worker threads and a entry point function

Parameters

entry_point	entry point funtion of the ums
completion_queue↩_id	id of the completion queue used by the ums
n_cpu	cpu where this ums will be scheduled

Returns

int

4.4.3.10 UmsThreadYield()

```
int UmsThreadYield (
              void  )
```

called from a worker thread, it pauses the execution of the current thread and the UMS scheduler entry point is executed for determining the next thread to be scheduled

Returns

int

4.5 ums.h

Go to the documentation of this file.

```
/*
 * This file is part of the User Mode Thread Scheduling (Kernel Module).
 * Copyright (c) 2021 Tiziano Colagrossi.
 *
 * This program 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, version 3.
 *
 * This program 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 this program. If not, see <http://www.gnu.org/licenses/>.
 */

#define _GNU_SOURCE

#include "../kernel_module/shared.h"
#include "list.h"
#include <stdlib.h>
#include <semaphore.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/syscall.h>
#include <time.h>

typedef void *(*worker_job)(void *);
typedef void *(*ums_entry_point)(void *);

typedef int bool;
#define TRUE  1
#define FALSE 0

#define GENERAL_UMS_ERROR -1
#define EXIT_UMS_MOD -2

#define UMS_PATH "/dev/ums"
#define MODULE_UMSLIB_LOG "[UMS LIB DEBUG]: "

#define UMSLIB_DEBUG

#define __F_APPEND "__append_new_worker_to_cq: "
#define F_APPEND "AppendToCompletionQueue: "
#define F_FLUSH "FlushCompletionQueue: "

#define INIT_UMS_ENTRY_STRUCT(X,E,I,O)  \
    ums_entry_info_t X = {        \
            .entry = E,     \
            .cq_id = I,     \
            .ret_value = 1, \
            .owner_pid = 0  \
            }


typedef struct buff_cq{
        int pids[COMPLETION_QUEUE_BUFF];
} dequeued_cq_t;

typedef struct cq_list_item
{
    int                   id;
    int              used_by;
    ums_cq_param_t   cq_item;
    struct list_head    list;
} cq_list_item_t;

typedef struct worker_thread_job_info {
    worker_job           job;
    void *      args_routine;
    int              pid;
} worker_thread_job_info_t;

typedef struct ums_entry_info {
    ums_entry_point    entry;
    int                cq_id;
    int            owner_pid;
    int            ret_value;
} ums_entry_info_t;

typedef struct pthread_entry {
    pthread_t        tid;
    struct list_head    list;
} pthread_entry_t;

typedef struct ums_scheduler{
    pthread_t * ums_threads_list;
    int         n_cpu;
    int cq_id;
} ums_t;

//Functions exported to user
int CreateNewWorker(worker_job job_to_perform, void * job_args);

int UmsThreadYield(void);
int ExecuteUmsThread(unsigned worker_id);

pthread_t UMS_thread_create(ums_entry_point entry_point, int completion_queue_id, int n_cpu);

ums_t * EnterUmsSchedulingMode(ums_entry_point entry_point, int completion_queue_id);
void ExitFromUmsSchedulingMode(ums_t * ums);

int CreateCompletionQueue(void);
int AppendToCompletionQueue(int completion_queue_id, int worker_pid);
int FlushCompletionQueue(int completion_queue_id);
int DequeueUmsCompletionListItems(dequeued_cq_t * return_cq);

void resetUMSFlag(void);
```

Index