User Mode thread Scheduling (user library)

Generated by Doxygen 1.9.3

1	Data Structure Index	1
	1.1 Data Structures	1
2	File Index	3
	2.1 File List	3
3	Data Structure Documentation	5
	3.1 buff_cq Struct Reference	5
	3.1.1 Detailed Description	5
	3.1.2 Field Documentation	5
	3.1.2.1 pids	5
	3.2 cq_list_item Struct Reference	6
	3.2.1 Field Documentation	6
	3.2.1.1 cq_item	6
	3.2.1.2 id	6
	3.2.1.3 list	6
	3.2.1.4 used_by	6
	3.3 list_head Struct Reference	7
	3.4 pthread_entry Struct Reference	7
	3.4.1 Field Documentation	7
	3.4.1.1 list	7
	3.4.1.2 tid	7
	3.5 ums_entry_info Struct Reference	7
	3.5.1 Field Documentation	8
	3.5.1.1 cq_id	8
	3.5.1.2 entry	8
	3.5.1.3 owner_pid	8
	3.5.1.4 ret_value	8
	3.6 ums_scheduler Struct Reference	8
	3.7 worker_thread_job_info Struct Reference	9
	3.7.1 Field Documentation	9
	3.7.1.1 args_routine	9
	3.7.1.2 job	9
	3.7.1.3 pid	9
4	File Documentation	11
	4.1 list.h File Reference	11
	4.1.1 Detailed Description	12
	4.1.2 Macro Definition Documentation	12
	4.1.2.1 container_of	12
	4.1.2.2 list_entry	12
	4.1.2.3 list_entry_is_head	12
	4.1.2.4 list_first_entry	13

4.1.2.5 list_for_each	 . 13
4.1.2.6 list_for_each_continue	 . 13
4.1.2.7 list_for_each_entry	 . 14
4.1.2.8 list_for_each_entry_continue	 . 14
4.1.2.9 list_for_each_entry_continue_reverse	 . 15
4.1.2.10 list_for_each_entry_from	 . 15
4.1.2.11 list_for_each_entry_from_reverse	 . 15
4.1.2.12 list_for_each_entry_reverse	 . 16
4.1.2.13 list_for_each_entry_safe	 . 16
4.1.2.14 list_for_each_entry_safe_continue	 . 17
4.1.2.15 list_for_each_entry_safe_from	 . 17
4.1.2.16 list_for_each_entry_safe_reverse	 . 18
4.1.2.17 list_for_each_prev	 . 18
4.1.2.18 list_for_each_prev_safe	 . 18
4.1.2.19 list_for_each_safe	 . 19
4.1.2.20 list_last_entry	 . 19
4.1.2.21 list_next_entry	 . 20
4.1.2.22 list_prepare_entry	 . 20
4.1.2.23 list_prev_entry	 . 20
4.1.2.24 list_safe_reset_next	 . 21
4.2 list.h	 . 21
4.3 ums.c File Reference	 . 26
4.3.1 Detailed Description	 . 27
4.3.2 Function Documentation	 . 27
4.3.2.1default_entry_point()	 . 27
4.3.2.2ums_entry_point_wrapper()	 . 27
4.3.2.3 AppendToCompletionQueue()	 . 28
4.3.2.4 CreateCompletionQueue()	 . 28
4.3.2.5 CreateNewWorker()	 . 28
4.3.2.6 DequeueUmsCompletionListItems()	 . 29
4.3.2.7 EnterUmsSchedulingMode()	 . 29
4.3.2.8 ExecuteUmsThread()	 . 30
4.3.2.9 ExitFromUmsSchedulingMode()	 . 30
4.3.2.10 FlushCompletionQueue()	 . 30
4.3.2.11 UMS_thread_create()	 . 31
4.3.2.12 UmsThreadYield()	 . 31
4.4 ums.h File Reference	 . 31
4.4.1 Detailed Description	 . 33
4.4.2 Macro Definition Documentation	 . 33
4.4.2.1 INIT_UMS_ENTRY_STRUCT	 . 33
4.4.3 Function Documentation	 . 33
4.4.3.1 AppendToCompletionQueue()	 . 33

Index		39
4.5 ums.h		37
	4.4.3.10 UmsThreadYield()	37
	4.4.3.9 UMS_thread_create()	36
	4.4.3.8 FlushCompletionQueue()	36
	4.4.3.7 ExitFromUmsSchedulingMode()	36
	4.4.3.6 ExecuteUmsThread()	35
	4.4.3.5 EnterUmsSchedulingMode()	35
	4.4.3.4 DequeueUmsCompletionListItems()	34
	4.4.3.3 CreateNewWorker()	34
	4.4.3.2 CreateCompletionQueue()	34

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

uff_cq
Contain the biffer used for the operation of the completion queue
q_list_item
st_head
hread_entry
ms_entry_info
ms_scheduler
orker thread job info

2 Data Structure Index

Chapter 2

File Index

2.1 File List

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

list.h		
	This file is the kernel implementation of the list is from Linux Kernel (include/linux/list.h)	11
ums.c		
	This file contains main definiton and function for the ums user library	26
ums.h		
	This file is the header of the user library	31

File Index

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 * nextstruct 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

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.
Ī	member	the name of the member within the struct.

4.1.2.2 list_entry

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(
```

4.1 list.h File Reference

```
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

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

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

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.	the head for your list.
member	the name of the list_head within the struct.

4.1.2.8 list for each entry continue

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.

4.1 list.h File Reference

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

4.1.2.9 list_for_each_entry_continue_reverse

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

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

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

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 \ast 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

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

4.1 list.h File Reference 17

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,
              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.
General bey Don't sename 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

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

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

Value:

4.1 list.h File Reference

```
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

Value:

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

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

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

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

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.

4.2 list.h 21

4.1.2.24 list_safe_reset_next

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.

```
9 /* SPDX-License-Identifier: GPL-2.0 */
10 #ifndef _LINUX_LIST_H
11 #define _LINUX_LIST_H
12 #include <stddef.h>
1.3
14 /*
15 * Circular doubly linked list implementation.
    \star Some of the internal functions ("__xxx") are useful when
18 \star manipulating whole lists rather than single entries, as
19 \,\star\, sometimes we already know the next/prev entries and we can 20 \,\star\, generate better code by using them directly rather than
21 * using the generic single-entry routines.
24 #ifndef offsetof
25 \#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
26 #endif
28 #ifndef container_of
36 #define container_of(ptr, type, member) ({
        (type *)((char *)ptr - offsetof(type, member)); })
38 #endif
39
40 struct list head {
41 struct list_head *next, *prev;
42 };
43
44 #define LIST_HEAD_INIT(name) { &(name), &(name) }
4.5
46 #define LIST HEAD(name) \
       struct list_head name = LIST_HEAD_INIT(name)
48
56 static inline void INIT_LIST_HEAD(struct list_head *list)
58
            list->next = list;
59
            list->prev = list;
60
        }while(0);
62 }
64 /*
65 * Insert a new entry between two known consecutive entries.
66 *
67 * This is only for internal list manipulation where we know
68 * the prev/next entries already!
```

```
70 static inline void __list_add(struct list_head *new,
71
                       struct list_head *prev,
72
                       struct list_head *next)
73 {
74
75
       next->prev = new;
76
        new->next = next;
77
        new->prev = prev;
78
       prev->next = new;
79 }
80
89 static inline void list_add(struct list_head *new, struct list_head *head)
90 {
91
        __list_add(new, head, head->next);
92 }
93
94
103 static inline void list_add_tail(struct list_head *new, struct list_head *head)
104 {
105
         __list_add(new, head->prev, head);
106 }
107
108 /*
109 * Delete a list entry by making the prev/next entries
110 * point to each other.
111 *
112 \,\,\star\,\, This is only for internal list manipulation where we know
113 * the prev/next entries already!
114 */
115 static inline void __list_del(struct list_head * prev, struct list_head * next)
116 {
117
         next->prev = prev;
118
           prev->next = next;
119 }
120
121 /*
122 * Delete a list entry and clear the 'prev' pointer.
123
124 \, \star This is a special-purpose list clearing method used in the networking code
125 * for lists allocated as per-cpu, where we don't want to incur the extra
126 * WRITE_ONCE() overhead of a regular list_del_init(). The code that uses this
127 * needs to check the node 'prev' pointer instead of calling list_empty().
128 */
129 static inline void __list_del_clearprev(struct list_head *entry)
130 {
131
           _list_del(entry->prev, entry->next);
132
         entry->prev = NULL;
133 }
134
135 static inline void __list_del_entry(struct list_head *entry)
136 {
137
138
         __list_del(entry->prev, entry->next);
139 }
140
147 static inline void list_del(struct list_head *entry)
148 {
149
          _list_del_entry(entry);
        entry->next = (void *)0;
entry->prev = (void *)0;
150
1.5.1
152 }
153
161 static inline void list_replace(struct list_head *old,
162
                      struct list_head *new)
163 {
164
        new->next = old->next;
        new->next->prev = new;
165
166
        new->prev = old->prev;
167
        new->prev->next = new;
168 }
169
177 static inline void list_replace_init(struct list_head *old,
178
                            struct list_head *new)
179 {
180
         list_replace(old, new);
181
         INIT_LIST_HEAD(old);
182 }
183
189 static inline void list_swap(struct list_head *entry1, 190 struct list_head *entry2)
191 {
192
         struct list_head *pos = entry2->prev;
193
194
        list_del(entry2);
195
         list_replace(entry1, entry2);
196
         if (pos == entrv1)
```

4.2 list.h 23

```
197
            pos = entry2;
198
        list_add(entry1, pos);
199 }
200
205 static inline void list del init(struct list head *entry)
206 {
          _list_del_entry(entry);
208
        INIT_LIST_HEAD(entry);
209 }
210
216 static inline void list_move(struct list_head *list, struct list_head *head)
217 {
          _list_del_entry(list);
218
219
        list_add(list, head);
220 }
221
227 static inline void list_move_tail(struct list_head *list,
228
                       struct list head *head)
229 {
230
          _list_del_entry(list);
231
        list_add_tail(list, head);
232 }
2.3.3
243 static inline void list_bulk_move_tail(struct list_head *head,
244
                            struct list_head *first,
245
                            struct list_head *last)
246 {
        first->prev->next = last->next;
last->next->prev = first->prev;
247
248
249
250
        head->prev->next = first;
251
        first->prev = head->prev;
252
        last->next = head;
head->prev = last;
253
2.54
255 }
256
262 static inline int list_is_first(const struct list_head *list,
263
                        const struct list_head *head)
264 {
265
        return list->prev == head;
266 }
2.67
273 static inline int list_is_last(const struct list_head *list,
274
                    const struct list_head *head)
275 {
276
        return list->next == head;
277 }
278
283 static inline int list_empty(const struct list_head *head)
284 {
285
        return head->next == head;
286 }
287
292 static inline void list_rotate_left(struct list_head *head)
293 {
294
        struct list_head *first;
295
296
        if (!list_empty(head)) {
297
             first = head->next;
298
            list_move_tail(first, head);
299
300 }
301
309 static inline void list_rotate_to_front(struct list_head *list,
310
                         struct list_head *head)
311 {
312
313
        * Deletes the list head from the list denoted by @head and
         * places it as the tail of @list, this effectively rotates the
314
315
         * list so that @list is at the front.
316
317
        list_move_tail(head, list);
318 }
319
324 static inline int list_is_singular(const struct list_head *head)
325 {
326
        return !list_empty(head) && (head->next == head->prev);
327 }
328
329 static inline void __list_cut_position(struct list_head *list,
330
            struct list_head *head, struct list_head *entry)
331 {
332
        struct list_head *new_first = entry->next;
333
        list->next = head->next;
        list->next->prev = list;
334
        list->prev = entry;
335
```

```
336
        entry->next = list;
337
        head->next = new_first;
338
        new_first->prev = head;
339 }
340
355 static inline void list_cut_position(struct list_head *list,
            struct list_head *head, struct list_head *entry)
357 {
358
        if (list_empty(head))
359
        if (list_is_singular(head) &&
360
            (head->next != entry && head != entry))
361
362
            return;
        if (entry == head)
363
364
            INIT_LIST_HEAD(list);
365
              _list_cut_position(list, head, entry);
366
367 }
368
383 static inline void list_cut_before(struct list_head *list,
384
                       struct list_head *head,
385
                        struct list_head *entry)
386 {
        if (head->next == entry) {
387
388
            INIT_LIST_HEAD(list);
            return;
390
391
        list->next = head->next;
392
        list->next->prev = list;
393
        list->prev = entry->prev;
394
        list->prev->next = list;
395
        head->next = entry;
396
        entry->prev = head;
397 }
398
399 static inline void __list_splice(const struct list_head *list,
                     struct list_head *prev,
struct list_head *next)
400
402 {
       struct list_head *first = list->next;
struct list_head *last = list->prev;
403
404
405
406
       first->prev = prev:
       prev->next = first;
407
408
409
        last->next = next;
410
        next->prev = last;
411 }
412
418 static inline void list_splice(const struct list_head *list,
                    struct list_head *head)
419
420 {
421
        if (!list_empty(list))
422
           __list_splice(list, head, head->next);
423 }
424
430 static inline void list_splice_tail(struct list_head *list,
431
                    struct list_head *head)
432 {
        if (!list_empty(list))
433
            __list_splice(list, head->prev, head);
434
435 }
436
444 static inline void list_splice_init(struct list_head *list,
445
                         struct list_head *head)
446 {
447
        if (!list_empty(list)) {
              _list_splice(list, head, head->next);
448
            INIT_LIST_HEAD(list);
449
450
        }
451 }
452
461 static inline void list_splice_tail_init(struct list_head *list,
                         struct list_head *head)
462
463 {
        if (!list_empty(list)) {
464
465
              _list_splice(list, head->prev, head);
466
            INIT_LIST_HEAD(list);
467
468 }
469
476 #define list_entry(ptr, type, member) \
477
        container_of(ptr, type, member)
478
487 #define list_first_entry(ptr, type, member) \setminus
488
        list_entry((ptr)->next, type, member)
489
```

4.2 list.h 25

```
498 #define list_last_entry(ptr, type, member) \
     list_entry((ptr)->prev, type, member)
500
506 #define list_next_entry(pos, member) \
507
      list_entry((pos)->member.next, typeof(*(pos)), member)
508
514 #define list_prev_entry(pos, member) \
      list_entry((pos)->member.prev, typeof(*(pos)), member)
515
516
522 #define list_for_each(pos, head) \
     for (pos = (head) ->next; pos != (head); pos = pos->next)
523
524
532 #define list for each continue(pos, head) \
     for (pos = pos->next; pos != (head); pos = pos->next)
533
534
540 #define list_for_each_prev(pos, head) \
541
      for (pos = (head) ->prev; pos != (head); pos = pos->prev)
542
549 #define list_for_each_safe(pos, n, head) \
     for (pos = (head)->next, n = pos->next; pos != (head); \
           pos = n, n = pos->next)
551
552
559 #define list_for_each_prev_safe(pos, n, head) \
      for (pos = (head) ->prev, n = pos->prev; \
    pos != (head); \
560
561
562
            pos = n, n = pos->prev)
563
570 #define list_entry_is_head(pos, head, member)
571
       (&pos->member == (head))
572
!list_entry_is_head(pos, head, member);
581
582
            pos = list_next_entry(pos, member))
583
590 #define list_for_each_entry_reverse(pos, head, member)
591
     for (pos = list_last_entry(head, typeof(*pos), member);
            !list_entry_is_head(pos, head, member);
593
            pos = list_prev_entry(pos, member))
594
603 #define list_prepare_entry(pos, head, member) \setminus
604
      ((pos) ? : list_entry(head, typeof(*pos), member))
605
615 #define list_for_each_entry_continue(pos, head, member)
616
      for (pos = list_next_entry(pos, member);
617
             !list_entry_is_head(pos, head, member);
618
            pos = list_next_entry(pos, member))
619
629 #define list_for_each_entry_continue_reverse(pos, head, member)
630
     for (pos = list_prev_entry(pos, member);
631
             !list_entry_is_head(pos, head, member);
632
            pos = list_prev_entry(pos, member))
633
642 #define list_for_each_entry_from(pos, head, member)
643
      for (; !list_entry_is_head(pos, head, member);
644
           pos = list next entry(pos, member))
655 #define list_for_each_entry_from_reverse(pos, head, member)
     for (; !list_entry_is_head(pos, head, member);
656
657
            pos = list_prev_entry(pos, member))
658
n = list_next_entry(pos, member);
669
            !list_entry_is_head(pos, head, member);
670
            pos = n, n = list_next_entry(n, member))
671
682 #define list_for_each_entry_safe_continue(pos, n, head, member)
683
       for (pos = list_next_entry(pos, member),
684
              = list_next_entry(pos, member);
685
             !list_entry_is_head(pos, head, member);
686
            pos = n, n = list_next_entry(n, member))
687
698 #define list_for_each_entry_safe_from(pos, n, head, member)
699
      for (n = list_next_entry(pos, member);
   !list_entry_is_head(pos, head, member);
700
            pos = n, n = list_next_entry(n, member))
701
702
713 #define list_for_each_entry_safe_reverse(pos, n, head, member)
714
       for (pos = list_last_entry(head, typeof(*pos), member),
715
           n = list_prev_entry(pos, member);
716
            !list_entry_is_head(pos, head, member);
            pos = n, n = list_prev_entry(n, member))
717
718
731 #define list_safe_reset_next(pos, n, member)
732
       n = list_next_entry(pos, member)
733
```

```
734
735 #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 cg 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

4.3 ums.c File Reference 27

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()

the default entry poont for the ums schedulers

Parameters

```
arguments
```

Returns

void*

4.3.2.2 __ums_entry_point_wrapper()

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

Parameters

```
args pointer to ums_entry_info_t struct
```

Returns

 $\mathsf{void} *$

4.3.2.3 AppendToCompletionQueue()

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()

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()

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.

4.3 ums.c File Reference 29

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()

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 of the completion queue used by the ums
_id	

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⊷	pid of the worker that will be executed
_id	

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()

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

Parameters

completion_queue←	id of the completion queue
id	

Returns

int

4.4 ums.h File Reference 31

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()

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)

and a entry point function

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

4.4 ums.h File Reference 33

• 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

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()

```
\begin{array}{c} \text{int CreateCompletionQueue (} \\ \text{void} \end{array} )
```

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()

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()

4.4 ums.h File Reference 35

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 of the completion queue used by the ums
_id	

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←	pid of the worker that will be executed
_id	

Returns

int

4.4.3.7 ExitFromUmsSchedulingMode()

Exit from UMS mode.

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

Parameters



4.4.3.8 FlushCompletionQueue()

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

Parameters

completion_queue←	id of the completion queue
_id	

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

4.5 ums.h 37

Returns

int

4.4.3.10 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

Returns

int

4.5 ums.h

Go to the documentation of this file.

```
2 \, \star 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.
  \star This program is distributed in the hope that it will be useful, but
10 * WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
   * General Public License for more details.
13 *
14 \, \star You should have received a copy of the GNU General Public License
1.5
   * along with this program. If not, see <a href="http://www.gnu.org/licenses/">http://www.gnu.org/licenses/</a>.
16 */
17
26 #define _GNU_SOURCE
28 #include "../kernel_module/shared.h"
29 #include "list.h"
30 #include <stdlib.h>
31 #include <semaphore.h>
32 #include <unistd.h>
33 #include <pthread.h>
34 #include <sys/types.h>
35 #include <sys/syscall.h>
36 #include <time.h>
38 typedef void *(*worker_job)(void *);
39 typedef void *(*ums_entry_point)(void *);
41 typedef int bool;
42 #define TRUE
43 #define FALSE 0
44
45 #define GENERAL_UMS_ERROR -1
46 #define EXIT_UMS_MOD -2
48 #define UMS_PATH "/dev/ums"
49 #define MODULE_UMSLIB_LOG "[UMS LIB DEBUG]: "
50
51 #define UMSLIB_DEBUG
53 #define __F_APPEND "__append_new_worker_to_cq: "
54 #define F_APPEND "AppendToCompletionQueue: "
55 #define F_FLUSH "FlushCompletionQueue: "
56
57 #define INIT_UMS_ENTRY_STRUCT(X, E, I, O) \
       ums_entry_info_t X = {
```

```
.entry = E,
               .cq_id = I,
.ret_value = 1,
61
               .owner_pid = 0
62
6.3
64
65
70 typedef struct buff_cq{
71
       int pids[COMPLETION_QUEUE_BUFF];
72 } dequeued_cq_t;
73
74 typedef struct cq_list_item
75 {
76
77
      int
                        used_by;
78
     ums_cq_param_t cq_item;
79  struct list_head
80 } cq_list_item_t;
                           list:
81
82 typedef struct worker_thread_job_info {
83
     worker_job
      void * args_routine;
84
8.5
      int
                    pid;
86 } worker_thread_job_info_t;
88 typedef struct ums_entry_info {
    ums_entry_point entry;
int cq_id;
int owner_pid;
89
90
     int
int
91
                     ret_value;
92
93 } ums_entry_info_t;
95 typedef struct pthread_entry {
   pthread_t tid;
struct list_head list;
96
97
98 } pthread_entry_t;
99
100 typedef struct ums_scheduler{
     pthread_t * ums_threads_list;
int n_cpu;
101
102
103
       int cq_id;
104 } ums_t;
105
106 //Functions exported to user
107 int CreateNewWorker(worker_job job_to_perform, void * job_args);
108
109 int UmsThreadYield(void);
110 int ExecuteUmsThread(unsigned worker_id);
111
112 pthread_t UMS_thread_create(ums_entry_point entry_point, int completion_queue_id, int n_cpu);
113
114 ums_t * EnterUmsSchedulingMode(ums_entry_point entry_point, int completion_queue_id);
115 void ExitFromUmsSchedulingMode(ums_t * ums);
116
117 int CreateCompletionQueue(void);
118 int AppendToCompletionQueue(int completion_queue_id, int worker_pid);
119 int FlushCompletionQueue(int completion_queue_id);
120 int DequeueUmsCompletionListItems(dequeued_cq_t * return_cq);
121
122 void resetUMSFlag(void);
```

Index

```
__default_entry_point
                                                         id
                                                              cq_list_item, 6
     ums.c, 27
                                                         INIT UMS_ENTRY_STRUCT
 _ums_entry_point_wrapper
                                                              ums.h, 33
     ums.c, 27
AppendToCompletionQueue
                                                         job
     ums.c, 28
                                                              worker_thread_job_info, 9
     ums.h, 33
                                                         list
args_routine
                                                              cq_list_item, 6
     worker_thread_job_info, 9
                                                              pthread_entry, 7
buff cq, 5
                                                         list.h, 11
     pids, 5
                                                              container_of, 12
                                                              list entry, 12
container_of
                                                              list_entry_is_head, 12
     list.h, 12
                                                              list first entry, 13
cq id
                                                              list for each, 13
     ums_entry_info, 8
                                                              list_for_each_continue, 13
cq_item
                                                              list_for_each_entry, 14
     cq_list_item, 6
                                                              list for each entry continue, 14
cq_list_item, 6
                                                              list_for_each_entry_continue_reverse, 15
     cq_item, 6
                                                              list_for_each_entry_from, 15
     id, 6
                                                              list_for_each_entry_from_reverse, 15
     list, 6
                                                              list for each entry reverse, 16
     used by, 6
                                                              list for each entry safe, 16
CreateCompletionQueue
                                                              list_for_each_entry_safe_continue, 17
     ums.c, 28
                                                              list_for_each_entry_safe_from, 17
     ums.h, 34
                                                              list for each entry safe reverse, 18
CreateNewWorker
                                                              list_for_each_prev, 18
     ums.c, 28
                                                              list_for_each_prev_safe, 18
     ums.h, 34
                                                              list_for_each_safe, 19
                                                              list_last_entry, 19
DequeueUmsCompletionListItems
                                                              list_next_entry, 19
     ums.c, 29
                                                              list_prepare_entry, 20
     ums.h, 34
                                                              list prev entry, 20
                                                              list safe reset next, 20
EnterUmsSchedulingMode
                                                         list entry
     ums.c, 29
                                                              list.h, 12
     ums.h, 35
                                                         list entry is head
entry
                                                              list.h, 12
     ums_entry_info, 8
                                                         list_first_entry
ExecuteUmsThread
                                                              list.h, 13
     ums.c, 29
                                                         list for each
     ums.h, 35
                                                              list.h, 13
ExitFromUmsSchedulingMode
                                                         list_for_each_continue
     ums.c, 30
                                                              list.h, 13
     ums.h, 35
                                                         list for each entry
                                                              list.h, 14
FlushCompletionQueue
                                                         list_for_each_entry_continue
     ums.c, 30
                                                              list.h, 14
     ums.h, 36
```

40 INDEX

list_for_each_entry_continue_reverse list.h, 15 list_for_each_entry_from	ExecuteUmsThread, 29 ExitFromUmsSchedulingMode, 30 FlushCompletionQueue, 30
list.h, 15	UMS_thread_create, 30
list_for_each_entry_from_reverse	UmsThreadYield, 31
list.h, 15	ums.h, 31
list_for_each_entry_reverse	AppendToCompletionQueue, 33
list.h, 16	CreateCompletionQueue, 34
list_for_each_entry_safe	CreateNewWorker, 34
list.h, 16	DequeueUmsCompletionListItems, 34
list_for_each_entry_safe_continue	EnterUmsSchedulingMode, 35
list.h, 17	ExecuteUmsThread, 35
list_for_each_entry_safe_from	ExitFromUmsSchedulingMode, 35
list.h, 17	FlushCompletionQueue, 36
list_for_each_entry_safe_reverse	INIT_UMS_ENTRY_STRUCT, 33
list.h, 18	UMS_thread_create, 36
list_for_each_prev	UmsThreadYield, 37
list.h, 18	ums_entry_info, 7
list_for_each_prev_safe	cq_id, 8
list.h, 18	entry, 8
list_for_each_safe	owner_pid, 8
list.h, 19	ret_value, 8
list_head, 7	ums_scheduler, 8
list_last_entry	UMS_thread_create
list.h, 19	ums.c, 30
list_next_entry	ums.h, 36
list.h, 19	UmsThreadYield
list_prepare_entry	ums.c, 31
list.h, 20	ums.h, 37
list_prev_entry	used_by
list.h, 20	cq_list_item, 6
list_safe_reset_next	
list.h, 20	worker_thread_job_info, 9
	args_routine, 9
owner_pid	job, 9
ums_entry_info, 8	pid, 9
nid	
pid	
worker_thread_job_info, 9	
pids	
buff_cq, 5	
pthread_entry, 7	
list, 7	
tid, 7	
ret value	
ums_entry_info, 8	
ums_emy_mo, o	
tid	
pthread_entry, 7	
ums.c, 26	
default_entry_point, 27	
ums_entry_point_wrapper, 27	
AppendToCompletionQueue, 28	
CreateCompletionQueue, 28	
CreateNewWorker, 28	
DequeueUmsCompletionListItems, 29	
Enterl ImsSchedulingMode 29	