

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
4.3.2.1 __default_entry_point()	27
4.3.2.2 __ums_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

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

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

buff_cq	Contain the biffer used for the operation of the completion queue	5
cq_list_item	6
list_head	7
pthread_entry	7
ums_entry_info	7
ums_scheduler	8
worker_thread_job_info	9

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

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
```

pointer to the job (function) 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

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

4.1.2.2 list_entry

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

list_entry - get the struct for this entry

Parameters

<i>ptr</i>	the &struct list_head pointer.
<i>type</i>	the type of the struct this is embedded in.
<i>member</i>	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

<i>pos</i>	the type * to cursor
<i>head</i>	the head for your list.
<i>member</i>	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

<i>ptr</i>	the list head to take the element from.
<i>type</i>	the type of the struct this is embedded in.
<i>member</i>	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

<i>pos</i>	the &struct list_head to use as a loop cursor.
<i>head</i>	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

<i>pos</i>	the &struct list_head to use as a loop cursor.
<i>head</i>	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

<i>pos</i>	the type * to use as a loop cursor.
<i>head</i>	the head for your list.
<i>member</i>	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

<i>pos</i>	the type * to use as a loop cursor.
<i>head</i>	the head for your list.
<i>member</i>	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

<i>pos</i>	the type * to use as a loop cursor.
<i>head</i>	the head for your list.
<i>member</i>	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

<i>pos</i>	the type * to use as a loop cursor.
<i>head</i>	the head for your list.
<i>member</i>	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

<i>pos</i>	the type * to use as a loop cursor.
<i>head</i>	the head for your list.
<i>member</i>	the name of the <code>list_head</code> 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

<i>pos</i>	the type * to use as a loop cursor.
<i>head</i>	the head for your list.
<i>member</i>	the name of the <code>list_head</code> 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

<i>pos</i>	the type * to use as a loop cursor.
<i>n</i>	another type * to use as temporary storage
<i>head</i>	the head for your list.
<i>member</i>	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

<i>pos</i>	the type * to use as a loop cursor.
<i>n</i>	another type * to use as temporary storage
<i>head</i>	the head for your list.
<i>member</i>	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

<i>pos</i>	the type * to use as a loop cursor.
<i>n</i>	another type * to use as temporary storage
<i>head</i>	the head for your list.
<i>member</i>	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

<i>pos</i>	the type * to use as a loop cursor.
<i>n</i>	another type * to use as temporary storage
<i>head</i>	the head for your list.
<i>member</i>	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

<i>pos</i>	the &struct list_head to use as a loop cursor.
<i>head</i>	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

<i>pos</i>	the &struct list_head to use as a loop cursor.
<i>n</i>	another &struct list_head to use as temporary storage
<i>head</i>	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

<i>pos</i>	the &struct list_head to use as a loop cursor.
<i>n</i>	another &struct list_head to use as temporary storage
<i>head</i>	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

<i>ptr</i>	the list head to take the element from.
<i>type</i>	the type of the struct this is embedded in.
<i>member</i>	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

<i>pos</i>	the type * to cursor
<i>member</i>	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

<i>pos</i>	the type * to use as a start point
<i>head</i>	the head of the list
<i>member</i>	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

<i>pos</i>	the type * to cursor
<i>member</i>	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

<i>pos</i>	the loop cursor used in the list_for_each_entry_safe loop
<i>n</i>	temporary storage used in list_for_each_entry_safe
<i>member</i>	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.](#)

```
1
2
3 9 /* SPDX-License-Identifier: GPL-2.0 */
4
5 10 #ifndef _LINUX_LIST_H
6
7 11 #define _LINUX_LIST_H
8
9 12 #include <stddef.h>
10
11 13
12 14 /*
13
14 15  * Circular doubly linked list implementation.
15
16 16  *
17 17  * Some of the internal functions ("__xxx") are useful when
18 18  * manipulating whole lists rather than single entries, as
19 19  * sometimes we already know the next/prev entries and we can
20 20  * generate better code by using them directly rather than
21 21  * using the generic single-entry routines.
22 22  */
23
24 23
25 24 #ifndef offsetof
26
27 25 #define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
28
29 26 #endif
30
31 27
32 28 #ifndef container_of
33
34 36 #define container_of(ptr, type, member) ({          \
35
36 37     (type *)((char *)ptr - offsetof(type, member)); \
37
38 38 #endif
39
40 39
41 40 struct list_head {
42
43 41     struct list_head *next, *prev;
44
45 42 };
46
47 43
48 44 #define LIST_HEAD_INIT(name) { &(name), &(name) }
49
50 45
51 46 #define LIST_HEAD(name) \
52 47     struct list_head name = LIST_HEAD_INIT(name)
53
54 48
55 56 static inline void INIT_LIST_HEAD(struct list_head *list)
56
57 57 {
58
59 58     do{
60
61 59         list->next = list;
62
63 60         list->prev = list;
64
65 61     }while(0);
66
67 62 }
68
69 63
70 64 /*
71
72 65  * Insert a new entry between two known consecutive entries.
73
74 66  *
75 67  * This is only for internal list manipulation where we know
76 68  * the prev/next entries already!
```

```

69  */
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
81  static inline void list_add(struct list_head *new, struct list_head *head)
82  {
83      __list_add(new, head, head->next);
84  }
85
86  static inline void list_add_tail(struct list_head *new, struct list_head *head)
87  {
88      __list_add(new, head->prev, head);
89  }
90
91  /*
92   * Delete a list entry by making the prev/next entries
93   * point to each other.
94   *
95   * This is only for internal list manipulation where we know
96   * the prev/next entries already!
97   */
98  static inline void __list_del(struct list_head * prev, struct list_head * next)
99  {
100     next->prev = prev;
101     prev->next = next;
102 }
103
104 /*
105  * Delete a list entry and clear the 'prev' pointer.
106  *
107  * This is a special-purpose list clearing method used in the networking code
108  * for lists allocated as per-cpu, where we don't want to incur the extra
109  * WRITE_ONCE() overhead of a regular list_del_init(). The code that uses this
110  * needs to check the node 'prev' pointer instead of calling list_empty().
111  */
112 static inline void __list_del_clearprev(struct list_head *entry)
113 {
114     __list_del(entry->prev, entry->next);
115     entry->prev = NULL;
116 }
117
118 static inline void __list_del_entry(struct list_head *entry)
119 {
120     __list_del(entry->prev, entry->next);
121 }
122
123 static inline void list_del(struct list_head *entry)
124 {
125     __list_del_entry(entry);
126     entry->next = (void *)0;
127     entry->prev = (void *)0;
128 }
129
130 static inline void list_replace(struct list_head *old,
131                                struct list_head *new)
132 {
133     new->next = old->next;
134     new->next->prev = new;
135     new->prev = old->prev;
136     new->prev->next = new;
137 }
138
139 static inline void list_replace_init(struct list_head *old,
140                                     struct list_head *new)
141 {
142     list_replace(old, new);
143     INIT_LIST_HEAD(old);
144 }
145
146 static inline void list_swap(struct list_head *entry1,
147                              struct list_head *entry2)
148 {
149     struct list_head *pos = entry2->prev;
150
151     list_del(entry2);
152     list_replace(entry1, entry2);
153     if (pos == entry1)

```

```

197     pos = entry2;
198     list_add(entry1, pos);
199 }
200
201 static inline void list_del_init(struct list_head *entry)
202 {
203     __list_del_entry(entry);
204     INIT_LIST_HEAD(entry);
205 }
206
207 static inline void list_move(struct list_head *list, struct list_head *head)
208 {
209     __list_del_entry(list);
210     list_add(list, head);
211 }
212
213 static inline void list_move_tail(struct list_head *list,
214                                   struct list_head *head)
215 {
216     __list_del_entry(list);
217     list_add_tail(list, head);
218 }
219
220 static inline void list_bulk_move_tail(struct list_head *head,
221                                       struct list_head *first,
222                                       struct list_head *last)
223 {
224     first->prev->next = last->next;
225     last->next->prev = first->prev;
226
227     head->prev->next = first;
228     first->prev = head->prev;
229
230     last->next = head;
231     head->prev = last;
232 }
233
234 static inline int list_is_first(const struct list_head *list,
235                                const struct list_head *head)
236 {
237     return list->prev == head;
238 }
239
240 static inline int list_is_last(const struct list_head *list,
241                                const struct list_head *head)
242 {
243     return list->next == head;
244 }
245
246 static inline int list_empty(const struct list_head *head)
247 {
248     return head->next == head;
249 }
250
251 static inline void list_rotate_left(struct list_head *head)
252 {
253     struct list_head *first;
254
255     if (!list_empty(head)) {
256         first = head->next;
257         list_move_tail(first, head);
258     }
259 }
260
261 static inline void list_rotate_to_front(struct list_head *list,
262                                         struct list_head *head)
263 {
264     /*
265      * Deletes the list head from the list denoted by @head and
266      * places it as the tail of @list, this effectively rotates the
267      * list so that @list is at the front.
268      */
269     list_move_tail(head, list);
270 }
271
272 static inline int list_is_singular(const struct list_head *head)
273 {
274     return !list_empty(head) && (head->next == head->prev);
275 }
276
277 static inline void __list_cut_position(struct list_head *list,
278                                       struct list_head *head, struct list_head *entry)
279 {
280     struct list_head *new_first = entry->next;
281     list->next = head->next;
282     list->next->prev = list;
283     list->prev = entry;

```

```

336     entry->next = list;
337     head->next = new_first;
338     new_first->prev = head;
339 }
340
341 static inline void list_cut_position(struct list_head *list,
342     struct list_head *head, struct list_head *entry)
343 {
344     if (list_empty(head))
345         return;
346     if (list_is_singular(head) &&
347         (head->next != entry && head != entry))
348         return;
349     if (entry == head)
350         INIT_LIST_HEAD(list);
351     else
352         __list_cut_position(list, head, entry);
353 }
354
355 static inline void list_cut_before(struct list_head *list,
356     struct list_head *head,
357     struct list_head *entry)
358 {
359     if (head->next == entry) {
360         INIT_LIST_HEAD(list);
361         return;
362     }
363     list->next = head->next;
364     list->next->prev = list;
365     list->prev = entry->prev;
366     list->prev->next = list;
367     head->next = entry;
368     entry->prev = head;
369 }
370
371 static inline void __list_splice(const struct list_head *list,
372     struct list_head *prev,
373     struct list_head *next)
374 {
375     struct list_head *first = list->next;
376     struct list_head *last = list->prev;
377
378     first->prev = prev;
379     prev->next = first;
380
381     last->next = next;
382     next->prev = last;
383 }
384
385 static inline void list_splice(const struct list_head *list,
386     struct list_head *head)
387 {
388     if (!list_empty(list))
389         __list_splice(list, head, head->next);
390 }
391
392 static inline void list_splice_tail(struct list_head *list,
393     struct list_head *head)
394 {
395     if (!list_empty(list))
396         __list_splice(list, head->prev, head);
397 }
398
399 static inline void list_splice_init(struct list_head *list,
400     struct list_head *head)
401 {
402     if (!list_empty(list)) {
403         __list_splice(list, head, head->next);
404         INIT_LIST_HEAD(list);
405     }
406 }
407
408 static inline void list_splice_tail_init(struct list_head *list,
409     struct list_head *head)
410 {
411     if (!list_empty(list)) {
412         __list_splice(list, head->prev, head);
413         INIT_LIST_HEAD(list);
414     }
415 }
416
417 #define list_entry(ptr, type, member) \
418     container_of(ptr, type, member)
419
420 #define list_first_entry(ptr, type, member) \
421     list_entry((ptr)->next, type, member)
422

```

```

498 #define list_last_entry(ptr, type, member) \
499     list_entry((ptr)->prev, type, member)
500
501 #define list_next_entry(pos, member) \
502     list_entry((pos)->member.next, typeof(*(pos)), member)
503
504 #define list_prev_entry(pos, member) \
505     list_entry((pos)->member.prev, typeof(*(pos)), member)
506
507 #define list_for_each(pos, head) \
508     for (pos = (head)->next; pos != (head); pos = pos->next)
509
510 #define list_for_each_continue(pos, head) \
511     for (pos = pos->next; pos != (head); pos = pos->next)
512
513 #define list_for_each_prev(pos, head) \
514     for (pos = (head)->prev; pos != (head); pos = pos->prev)
515
516 #define list_for_each_safe(pos, n, head) \
517     for (pos = (head)->next, n = pos->next; pos != (head); \
518         pos = n, n = pos->next)
519
520 #define list_for_each_prev_safe(pos, n, head) \
521     for (pos = (head)->prev, n = pos->prev; \
522         pos != (head); \
523         pos = n, n = pos->prev)
524
525 #define list_entry_is_head(pos, head, member) \
526     (&pos->member == (head))
527
528 #define list_for_each_entry(pos, head, member) \
529     for (pos = list_first_entry(head, typeof(*pos), member); \
530         !list_entry_is_head(pos, head, member); \
531         pos = list_next_entry(pos, member))
532
533 #define list_for_each_entry_reverse(pos, head, member) \
534     for (pos = list_last_entry(head, typeof(*pos), member); \
535         !list_entry_is_head(pos, head, member); \
536         pos = list_prev_entry(pos, member))
537
538 #define list_prepare_entry(pos, head, member) \
539     ((pos) ? : list_entry(head, typeof(*pos), member))
540
541 #define list_for_each_entry_continue(pos, head, member) \
542     for (pos = list_next_entry(pos, member); \
543         !list_entry_is_head(pos, head, member); \
544         pos = list_next_entry(pos, member))
545
546 #define list_for_each_entry_continue_reverse(pos, head, member) \
547     for (pos = list_prev_entry(pos, member); \
548         !list_entry_is_head(pos, head, member); \
549         pos = list_prev_entry(pos, member))
550
551 #define list_for_each_entry_from(pos, head, member) \
552     for (; !list_entry_is_head(pos, head, member); \
553         pos = list_next_entry(pos, member))
554
555 #define list_for_each_entry_from_reverse(pos, head, member) \
556     for (; !list_entry_is_head(pos, head, member); \
557         pos = list_prev_entry(pos, member))
558
559 #define list_for_each_entry_safe(pos, n, head, member) \
560     for (pos = list_first_entry(head, typeof(*pos), member), \
561         n = list_next_entry(pos, member); \
562         !list_entry_is_head(pos, head, member); \
563         pos = n, n = list_next_entry(n, member))
564
565 #define list_for_each_entry_safe_continue(pos, n, head, member) \
566     for (pos = list_next_entry(pos, member), \
567         n = list_next_entry(pos, member); \
568         !list_entry_is_head(pos, head, member); \
569         pos = n, n = list_next_entry(n, member))
570
571 #define list_for_each_entry_safe_from(pos, n, head, member) \
572     for (n = list_next_entry(pos, member); \
573         !list_entry_is_head(pos, head, member); \
574         pos = n, n = list_next_entry(n, member))
575
576 #define list_for_each_entry_safe_reverse(pos, n, head, member) \
577     for (pos = list_last_entry(head, typeof(*pos), member), \
578         n = list_prev_entry(pos, member); \
579         !list_entry_is_head(pos, head, member); \
580         pos = n, n = list_prev_entry(n, member))
581
582 #define list_safe_reset_next(pos, n, member) \
583     n = list_next_entry(pos, member)
584
585

```

```

734
735 #endif

```

4.3 ums.c File Reference

This file contains main definition 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 point 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 number of cores in the computer) and will schedule the thread 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 before 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

<i>arguments</i>	
------------------	--

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

<i>args</i>	pointer to <code>ums_entry_info_t</code> 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 before it perform some check in order to see if the completion queue exist

Parameters

<i>completion_queue_id</i>	id of the completion queue where apped the worker
<i>worker_pid</i>	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

<i>job_to_perform</i>	job function of the worker thread
<i>job_args</i>	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

<i>return_cq</i>	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

<i>entry_point</i>	entry_point for the sceduler uf null will use the default
<i>completion_queue↵_id</i>	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

<i>worker_id</i>	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

<i>ums</i>	
------------	--

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

<i>completion_queue_id</i>	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

<i>entry_point</i>	entry point funtion of the ums
<i>completion_queue_id</i>	id of the completion queue used by the ums
<i>n_cpu</i>	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

<i>completion_queue↔ _id</i>	id of the completion queue where added the worker
<i>worker_pid</i>	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

<i>job_to_perform</i>	job function of the worker thread
<i>job_args</i>	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

<i>return_cq</i>	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

<i>entry_point</i>	entry_point for the sceduler uf null will use the default
<i>completion_queue↔_id</i>	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

<i>worker↔_id</i>	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

<i>ums</i>	
------------	--

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

<i>completion_queue↔ _id</i>	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

<i>entry_point</i>	entry point funtion of the ums
<i>completion_queue↔ _id</i>	id of the completion queue used by the ums
<i>n_cpu</i>	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.](#)

```
1 /*
2  * This file is part of the User Mode Thread Scheduling (Kernel Module).
3  * Copyright (c) 2021 Tiziano Colagrossi.
4  *
5  * This program is free software: you can redistribute it and/or modify
6  * it under the terms of the GNU General Public License as published by
7  * the Free Software Foundation, version 3.
8  *
9  * 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
12 * General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see <http://www.gnu.org/licenses/>.
16 */
17
26 #define _GNU_SOURCE
27
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>
37
38 typedef void *(*worker_job)(void *);
39 typedef void *(*ums_entry_point)(void *);
40
41 typedef int bool;
42 #define TRUE 1
43 #define FALSE 0
44
45 #define GENERAL_UMS_ERROR -1
46 #define EXIT_UMS_MOD -2
47
48 #define UMS_PATH "/dev/ums"
49 #define MODULE_UMSLIB_LOG "[UMS LIB DEBUG]: "
50
51 #define UMSLIB_DEBUG
52
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) \
58     ums_entry_info_t X = { \
```

```

59         .entry = E,      \
60         .cq_id = I,      \
61         .ret_value = 1,  \
62         .owner_pid = 0   \
63     }
64
65
66
67
68
69
70 typedef struct buff_cq{
71     int pids[COMPLETION_QUEUE_BUFF];
72 } dequeued_cq_t;
73
74 typedef struct cq_list_item
75 {
76     int id;
77     int used_by;
78     ums_cq_param_t cq_item;
79     struct list_head list;
80 } cq_list_item_t;
81
82 typedef struct worker_thread_job_info {
83     worker_job job;
84     void * args_routine;
85     int pid;
86 } worker_thread_job_info_t;
87
88 typedef struct ums_entry_info {
89     ums_entry_point entry;
90     int cq_id;
91     int owner_pid;
92     int ret_value;
93 } ums_entry_info_t;
94
95 typedef struct pthread_entry {
96     pthread_t tid;
97     struct list_head list;
98 } pthread_entry_t;
99
100 typedef struct ums_scheduler{
101     pthread_t * ums_threads_list;
102     int n_cpu;
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](#)
[ums.c, 27](#)
 - [__ums_entry_point_wrapper](#)
[ums.c, 27](#)
- [AppendToCompletionQueue](#)
 - [ums.c, 28](#)
 - [ums.h, 33](#)
- [args_routine](#)
 - [worker_thread_job_info, 9](#)
- [buff_cq, 5](#)
 - [pids, 5](#)
- [container_of](#)
 - [list.h, 12](#)
- [cq_id](#)
 - [ums_entry_info, 8](#)
- [cq_item](#)
 - [cq_list_item, 6](#)
- [cq_list_item, 6](#)
 - [cq_item, 6](#)
 - [id, 6](#)
 - [list, 6](#)
 - [used_by, 6](#)
- [CreateCompletionQueue](#)
 - [ums.c, 28](#)
 - [ums.h, 34](#)
- [CreateNewWorker](#)
 - [ums.c, 28](#)
 - [ums.h, 34](#)
- [DequeueUmsCompletionListItems](#)
 - [ums.c, 29](#)
 - [ums.h, 34](#)
- [EnterUmsSchedulingMode](#)
 - [ums.c, 29](#)
 - [ums.h, 35](#)
- [entry](#)
 - [ums_entry_info, 8](#)
- [ExecuteUmsThread](#)
 - [ums.c, 29](#)
 - [ums.h, 35](#)
- [ExitFromUmsSchedulingMode](#)
 - [ums.c, 30](#)
 - [ums.h, 35](#)
- [FlushCompletionQueue](#)
 - [ums.c, 30](#)
 - [ums.h, 36](#)
- [id](#)
 - [cq_list_item, 6](#)
- [INIT_UMS_ENTRY_STRUCT](#)
 - [ums.h, 33](#)
- [job](#)
 - [worker_thread_job_info, 9](#)
- [list](#)
 - [cq_list_item, 6](#)
 - [pthread_entry, 7](#)
- [list.h, 11](#)
 - [container_of, 12](#)
 - [list_entry, 12](#)
 - [list_entry_is_head, 12](#)
 - [list_first_entry, 13](#)
 - [list_for_each, 13](#)
 - [list_for_each_continue, 13](#)
 - [list_for_each_entry, 14](#)
 - [list_for_each_entry_continue, 14](#)
 - [list_for_each_entry_continue_reverse, 15](#)
 - [list_for_each_entry_from, 15](#)
 - [list_for_each_entry_from_reverse, 15](#)
 - [list_for_each_entry_reverse, 16](#)
 - [list_for_each_entry_safe, 16](#)
 - [list_for_each_entry_safe_continue, 17](#)
 - [list_for_each_entry_safe_from, 17](#)
 - [list_for_each_entry_safe_reverse, 18](#)
 - [list_for_each_prev, 18](#)
 - [list_for_each_prev_safe, 18](#)
 - [list_for_each_safe, 19](#)
 - [list_last_entry, 19](#)
 - [list_next_entry, 19](#)
 - [list_prepare_entry, 20](#)
 - [list_prev_entry, 20](#)
 - [list_safe_reset_next, 20](#)
- [list_entry](#)
 - [list.h, 12](#)
- [list_entry_is_head](#)
 - [list.h, 12](#)
- [list_first_entry](#)
 - [list.h, 13](#)
- [list_for_each](#)
 - [list.h, 13](#)
- [list_for_each_continue](#)
 - [list.h, 13](#)
- [list_for_each_entry](#)
 - [list.h, 14](#)
- [list_for_each_entry_continue](#)
 - [list.h, 14](#)

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