UMS library

Generated by Doxygen 1.9.2

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 2

File Index

2.1 File List

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

Chapter 3

Data Structure Documentation

3.1 cl_list Struct Reference

The list of completion lists.

`#include <ums_lib.h>`

Data Fields

- struct list_head **list**
- unsigned int cl_count

3.1.1 Detailed Description

The list of completion lists.

The purpose of this list is to store all completion lists created by the program

3.1.2 Field Documentation

3.1.2.1 cl_count

`unsigned int cl_list::cl_count`

The number of elements(completion lists) in the list

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

- ums_lib.h

3.2 completion_list Struct Reference

The completion list of worker threads.

`#include <ums_lib.h>`

Data Fields

- unsigned int id
- unsigned int worker_thread_count
- struct list_head **list**

3.2.1 Detailed Description

The completion list of worker threads.

This is a node in the cl_list.

3.2.2 Field Documentation

3.2.2.1 id

`unsigned int completion_list::id`

Unique id of the completion list

3.2.2.2 worker_thread_count

`unsigned int completion_list::worker_thread_count`

The number of worker threads in this completion list

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

- ums_lib.h

3.3 hlist_head Struct Reference

Data Fields

- struct hlist_node * **first**

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

- list.h

3.4 hlist_node Struct Reference

Data Fields

- struct hlist_node * **next**
- struct hlist_node ** **pprev**

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

- list.h

3.5 list_head Struct Reference

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

Data Fields

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

3.5.1 Detailed Description

Simple doubly linked list implementation.

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

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

- list.h

3.6 ums_thread Struct Reference

The ums thread(scheduler)

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

Data Fields

- unsigned int id
- pthread_t pt
- ums_thread_params_t * params
- struct list_head **list**

3.6.1 Detailed Description

The ums thread(scheduler)

This is a node in the ums_thread_list.

3.6.2 Field Documentation

3.6.2.1 id

```
unsigned int ums_thread::id
```

Unique id of the ums thread

3.6.2.2 params

```
ums_thread_params_t* ums_thread::params
```

ums_thread_params_t

3.6.2.3 pt

```
pthread_t ums_thread::pt
```

pthread which entered ums scheduling mode

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

- ums_lib.h

3.7 ums_thread_list Struct Reference

The list of ums threads(schedulers)

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

Data Fields

- struct list_head **list**
- unsigned int ums_thread_count

3.7.1 Detailed Description

The list of ums threads(schedulers)

The purpose of this list is to store all ums threads(schedulers) created by the program

3.7.2 Field Documentation

3.7.2.1 ums_thread_count

```
unsigned int ums_thread_list::ums_thread_count
```

The number of elements(ums threads) in the list

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

- ums_lib.h

3.8 worker_thread Struct Reference

The worker thread.

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

Data Fields

- unsigned int id
- worker_thread_params_t * params
- struct list_head **list**

3.8.1 Detailed Description

The worker thread.

This is a node in the worker_thread_list.

3.8.2 Field Documentation

3.8.2.1 id

```
unsigned int worker_thread::id
```

Unique id of the worker thread

3.8.2.2 params

```
worker_thread_params_t* worker_thread::params
```

worker_thread_params_t

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

- ums_lib.h

3.9 worker_thread_list Struct Reference

The list of worker threads.

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

Data Fields

- struct list_head **list**
- unsigned int worker_thread_count

3.9.1 Detailed Description

The list of worker threads.

The purpose of this list is to store all worker threads created by the program

3.9.2 Field Documentation

3.9.2.1 worker_thread_count

```
unsigned int worker_thread_list::worker_thread_count
```

The number of elements(worker threads) in the list

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

- ums_lib.h

Chapter 4

File Documentation

4.1 list.h File Reference

This file is the implementation of the list structure for user space environment: https://www.mcs.anl.↩gov/~kazutomo/list/list.h "I grub it from linux kernel source code and fix it for user space program. Of course, this is a GPL licensed header file. Here is a recipe to cook list.h for user space program.

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

Data Structures

- struct list_head
- struct hlist_head
- struct hlist_node

from other kernel headers

- #define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
- #define container_of(ptr, type, member)
- #define **LIST_POISON1** ((void *) 0x00100100)
- #define **LIST_POISON2** ((void *) 0x00200200)
- #define **LIST_HEAD_INIT**(name) { &(name), &(name) }
- #define **LIST_HEAD**(name) struct list_head name = LIST_HEAD_INIT(name)
- #define INIT_LIST_HEAD(ptr)
- #define list_entry(ptr, type, member) container_of(ptr, type, member)
- #define list_for_each(pos, head)
- #define __list_for_each(pos, head) for (pos = (head)->next; pos != (head); pos = pos->next)
- #define list_for_each_prev(pos, head)
- #define list_for_each_safe(pos, n, head)
- #define list_for_each_entry(pos, head, member)
- #define list_for_each_entry_reverse(pos, head, member)
- #define list_prepare_entry(pos, head, member) ((pos) ? : list_entry(head, typeof(*pos), member))
- #define list_for_each_entry_continue(pos, head, member)
- #define list_for_each_entry_safe(pos, n, head, member)
- #define list_for_each_entry_safe_continue(pos, n, head, member)

- #define list_for_each_entry_safe_reverse(pos, n, head, member)
- #define **HLIST_HEAD_INIT** { .first = NULL }
- #define **HLIST_HEAD**(name) struct hlist_head name = { .first = NULL }
- #define **INIT_HLIST_HEAD**(ptr) ((ptr)->first = NULL)
- #define **INIT_HLIST_NODE**(ptr) ((ptr)->next = NULL, (ptr)->pprev = NULL)
- #define **hlist_entry**(ptr, type, member) container_of(ptr,type,member)
- #define hlist_for_each(pos, head)
- #define hlist_for_each_safe(pos, n, head)
- #define hlist_for_each_entry(tpos, pos, head, member)
- #define hlist_for_each_entry_continue(tpos, pos, member)
- #define hlist_for_each_entry_from(tpos, pos, member)
- #define hlist_for_each_entry_safe(tpos, pos, n, head, member)

4.1.1 Detailed Description

This file is the implementation of the list structure for user space environment: https://www.mcs.anl.↩gov/~kazutomo/list/list.h "I grub it from linux kernel source code and fix it for user space program. Of course, this is a GPL licensed header file. Here is a recipe to cook list.h for user space program.

Copyright (C) 2021 Sultan Umarbaev name.sul27@gmail.com

This file is part of UMS implementation (Kernel Module).

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

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

You should have received a copy of the GNU General Public License along with UMS implementation (Kernel Module). If not, see http://www.gnu.org/licenses/.

1. copy list.h from linux/include/list.h
2. remove
   - #ifdef **KERNE** and its #endif
   - all #include line
   - prefetch() and rcu related functions
3. add macro offsetof() and container_of

- kazutomo@mcs.anl.gov"

Author

kazutomo@mcs.anl.gov

4.1.2 Macro Definition Documentation

4.1.2.1 __list_for_each

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

__list_for_each - iterate over a list @pos: the &struct list_head to use as a loop counter. @head: the head for your list.

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

4.1.2.2 container_of

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

Value:

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

Casts a member of a structure out to the containing structure

Parameters

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

4.1.2.3 hlist_for_each

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

Value:

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

4.1.2.4 hlist_for_each_entry

```
#define hlist_for_each_entry(
                tpos,
                pos,
```

```
            head,
            member )
```

Value:

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

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

4.1.2.5 hlist_for_each_entry_continue

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

Value:

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

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

4.1.2.6 hlist_for_each_entry_from

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

Value:

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

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

4.1.2.7 hlist_for_each_entry_safe

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

Value:

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

hlist_for_each_entry_safe - iterate over list of given type safe against removal of list entry @tpos: the type * to use as a loop counter. @pos: the &struct hlist_node to use as a loop counter.
: another &struct hlist_node to use as temporary storage @head: the head for your list. @member: the name of the hlist_node within the struct.

4.1.2.8 hlist_for_each_safe

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

Value:

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

4.1.2.9 INIT_LIST_HEAD

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

Value:

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

4.1.2.10 list_entry

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

list_entry - get the struct for this entry @ptr: the &struct list_head pointer. @type: the type of the struct this is embedded in. @member: the name of the list_struct within the struct.

4.1.2.11 list_for_each

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

Value:

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

list_for_each - iterate over a list @pos: the &struct list_head to use as a loop counter. @head: the head for your list.

4.1.2.12 list_for_each_entry

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

Value:

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

list_for_each_entry - iterate over list of given type @pos: the type * to use as a loop counter. @head: the head for your list. @member: the name of the list_struct within the struct.

4.1.2.13 list_for_each_entry_continue

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

Value:

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

list_for_each_entry_continue - iterate over list of given type continuing after existing point @pos: the type * to use as a loop counter. @head: the head for your list. @member: the name of the list_struct within the struct.

4.1.2.14 list_for_each_entry_reverse

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

Value:

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

list_for_each_entry_reverse - iterate backwards over list of given type. @pos: the type * to use as a loop counter. @head: the head for your list. @member: the name of the list_struct within the struct.

4.1.2.15 list_for_each_entry_safe

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

Value:

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

list_for_each_entry_safe - iterate over list of given type safe against removal of list entry @pos: the type * to use as a loop counter.
: another type * to use as temporary storage @head: the head for your list. @member: the name of the list_struct within the struct.

4.1.2.16 list_for_each_entry_safe_continue

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

Value:

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

list_for_each_entry_safe_continue - iterate over list of given type continuing after existing point safe against removal of list entry @pos: the type ∗ to use as a loop counter.
: another type ∗ to use as temporary storage @head: the head for your list. @member: the name of the list_struct within the struct.

4.1.2.17 list_for_each_entry_safe_reverse

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

Value:

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

list_for_each_entry_safe_reverse - iterate backwards over list of given type safe against removal of list entry @pos: the type ∗ to use as a loop counter.
: another type ∗ to use as temporary storage @head: the head for your list. @member: the name of the list_struct within the struct.

4.1.2.18 list_for_each_prev

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

Value:

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

list_for_each_prev - iterate over a list backwards @pos: the &struct list_head to use as a loop counter. @head: the head for your list.

4.1.2.19 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 @pos: the &struct list_head to use as a loop counter.
: another &struct list_head to use as temporary storage @head: the head for your list.

4.1.2.20 list_prepare_entry

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

list_prepare_entry - prepare a pos entry for use as a start point in list_for_each_entry_continue @pos: the type * to use as a start point @head: the head of the list @member: the name of the list_struct within the struct.

4.1.2.21 offsetof

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

Get offset of a member

4.2 list.h

Go to the documentation of this file.

```
1
41 #ifndef _LINUX_LIST_H
42 #define _LINUX_LIST_H
43 #include <stdio.h>
44
49
53 #define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
54
62 #define container_of(ptr, type, member) ({                      \
63         const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
64         (type *)( (char *)__mptr - offsetof(type,member) );})
68 /*
69  * These are non-NULL pointers that will result in page faults
70  * under normal circumstances, used to verify that nobody uses
71  * non-initialized list entries.
72  */
73 #define LIST_POISON1  ((void *) 0x00100100)
74 #define LIST_POISON2  ((void *) 0x00200200)
75
85 struct list_head {
86     struct list_head *next, *prev;
87 };
88
89 #define LIST_HEAD_INIT(name) { &(name), &(name) }
90
```

```
91 #define LIST_HEAD(name) \
92     struct list_head name = LIST_HEAD_INIT(name)
93
94 #define INIT_LIST_HEAD(ptr) do { \
95     (ptr)->next = (ptr); (ptr)->prev = (ptr); \
96 } while (0)
97
98 /*
99  * Insert a new entry between two known consecutive entries.
100  *
101  * This is only for internal list manipulation where we know
102  * the prev/next entries already!
103  */
104 static inline void __list_add(struct list_head *new,
105                   struct list_head *prev,
106                   struct list_head *next)
107 {
108     next->prev = new;
109     new->next = next;
110     new->prev = prev;
111     prev->next = new;
112 }
113
122 static inline void list_add(struct list_head *new, struct list_head *head)
123 {
124     __list_add(new, head, head->next);
125 }
126
135 static inline void list_add_tail(struct list_head *new, struct list_head *head)
136 {
137     __list_add(new, head->prev, head);
138 }
139
140
141 /*
142  * Delete a list entry by making the prev/next entries
143  * point to each other.
144  *
145  * This is only for internal list manipulation where we know
146  * the prev/next entries already!
147  */
148 static inline void __list_del(struct list_head * prev, struct list_head * next)
149 {
150     next->prev = prev;
151     prev->next = next;
152 }
153
160 static inline void list_del(struct list_head *entry)
161 {
162     __list_del(entry->prev, entry->next);
163     entry->next = LIST_POISON1;
164     entry->prev = LIST_POISON2;
165 }
166
167
168
173 static inline void list_del_init(struct list_head *entry)
174 {
175     __list_del(entry->prev, entry->next);
176     INIT_LIST_HEAD(entry);
177 }
178
184 static inline void list_move(struct list_head *list, struct list_head *head)
185 {
186         __list_del(list->prev, list->next);
187         list_add(list, head);
188 }
189
195 static inline void list_move_tail(struct list_head *list,
196                   struct list_head *head)
197 {
198         __list_del(list->prev, list->next);
199         list_add_tail(list, head);
200 }
201
206 static inline int list_empty(const struct list_head *head)
207 {
208     return head->next == head;
209 }
210
211 static inline void __list_splice(struct list_head *list,
212                  struct list_head *head)
213 {
214     struct list_head *first = list->next;
215     struct list_head *last = list->prev;
216     struct list_head *at = head->next;
217
```

```
218     first->prev = head;
219     head->next = first;
220
221     last->next = at;
222     at->prev = last;
223 }
224
230 static inline void list_splice(struct list_head *list, struct list_head *head)
231 {
232     if (!list_empty(list))
233         __list_splice(list, head);
234 }
235
243 static inline void list_splice_init(struct list_head *list,
244                     struct list_head *head)
245 {
246     if (!list_empty(list)) {
247         __list_splice(list, head);
248         INIT_LIST_HEAD(list);
249     }
250 }
251
258 #define list_entry(ptr, type, member) \
259     container_of(ptr, type, member)
260
267 #define list_for_each(pos, head) \
268   for (pos = (head)->next; pos != (head);   \
269         pos = pos->next)
270
281 #define __list_for_each(pos, head) \
282     for (pos = (head)->next; pos != (head); pos = pos->next)
283
289 #define list_for_each_prev(pos, head) \
290     for (pos = (head)->prev; prefetch(pos->prev), pos != (head); \
291             pos = pos->prev)
292
299 #define list_for_each_safe(pos, n, head) \
300     for (pos = (head)->next, n = pos->next; pos != (head); \
301         pos = n, n = pos->next)
302
309 #define list_for_each_entry(pos, head, member)              \
310     for (pos = list_entry((head)->next, typeof(*pos), member);  \
311          &pos->member != (head);                    \
312          pos = list_entry(pos->member.next, typeof(*pos), member))
313
320 #define list_for_each_entry_reverse(pos, head, member)          \
321     for (pos = list_entry((head)->prev, typeof(*pos), member);  \
322          &pos->member != (head);    \
323          pos = list_entry(pos->member.prev, typeof(*pos), member))
324
332 #define list_prepare_entry(pos, head, member) \
333     ((pos) ? : list_entry(head, typeof(*pos), member))
334
342 #define list_for_each_entry_continue(pos, head, member)         \
343     for (pos = list_entry(pos->member.next, typeof(*pos), member);  \
344          &pos->member != (head);    \
345          pos = list_entry(pos->member.next, typeof(*pos), member))
346
354 #define list_for_each_entry_safe(pos, n, head, member)          \
355     for (pos = list_entry((head)->next, typeof(*pos), member),  \
356         n = list_entry(pos->member.next, typeof(*pos), member); \
357          &pos->member != (head);                    \
358          pos = n, n = list_entry(n->member.next, typeof(*n), member))
359
368 #define list_for_each_entry_safe_continue(pos, n, head, member)         \
369     for (pos = list_entry(pos->member.next, typeof(*pos), member),      \
370         n = list_entry(pos->member.next, typeof(*pos), member);     \
371          &pos->member != (head);                    \
372          pos = n, n = list_entry(n->member.next, typeof(*n), member))
373
382 #define list_for_each_entry_safe_reverse(pos, n, head, member)      \
383     for (pos = list_entry((head)->prev, typeof(*pos), member),  \
384         n = list_entry(pos->member.prev, typeof(*pos), member); \
385          &pos->member != (head);                    \
386          pos = n, n = list_entry(n->member.prev, typeof(*n), member))
387
388
389
390
391 /*
392  * Double linked lists with a single pointer list head.
393  * Mostly useful for hash tables where the two pointer list head is
394  * too wasteful.
395  * You lose the ability to access the tail in O(1).
396  */
397
398 struct hlist_head {
```

```
399     struct hlist_node *first;
400 };
401
402 struct hlist_node {
403     struct hlist_node *next, **pprev;
404 };
405
406 #define HLIST_HEAD_INIT { .first = NULL }
407 #define HLIST_HEAD(name) struct hlist_head name = {  .first = NULL }
408 #define INIT_HLIST_HEAD(ptr) ((ptr)->first = NULL)
409 #define INIT_HLIST_NODE(ptr) ((ptr)->next = NULL, (ptr)->pprev = NULL)
410
411 static inline int hlist_unhashed(const struct hlist_node *h)
412 {
413     return !h->pprev;
414 }
415
416 static inline int hlist_empty(const struct hlist_head *h)
417 {
418     return !h->first;
419 }
420
421 static inline void __hlist_del(struct hlist_node *n)
422 {
423     struct hlist_node *next = n->next;
424     struct hlist_node **pprev = n->pprev;
425     *pprev = next;
426     if (next)
427         next->pprev = pprev;
428 }
429
430 static inline void hlist_del(struct hlist_node *n)
431 {
432     __hlist_del(n);
433     n->next = LIST_POISON1;
434     n->pprev = LIST_POISON2;
435 }
436
437
438 static inline void hlist_del_init(struct hlist_node *n)
439 {
440     if (n->pprev)  {
441         __hlist_del(n);
442         INIT_HLIST_NODE(n);
443     }
444 }
445
446 static inline void hlist_add_head(struct hlist_node *n, struct hlist_head *h)
447 {
448     struct hlist_node *first = h->first;
449     n->next = first;
450     if (first)
451         first->pprev = &n->next;
452     h->first = n;
453     n->pprev = &h->first;
454 }
455
456
457
458 /* next must be != NULL */
459 static inline void hlist_add_before(struct hlist_node *n,
460                     struct hlist_node *next)
461 {
462     n->pprev = next->pprev;
463     n->next = next;
464     next->pprev = &n->next;
465     *(n->pprev) = n;
466 }
467
468 static inline void hlist_add_after(struct hlist_node *n,
469                     struct hlist_node *next)
470 {
471     next->next = n->next;
472     n->next = next;
473     next->pprev = &n->next;
474
475     if(next->next)
476         next->next->pprev  = &next->next;
477 }
478
479
480
481 #define hlist_entry(ptr, type, member) container_of(ptr,type,member)
482
483 #define hlist_for_each(pos, head) \
484     for (pos = (head)->first; pos && ({ prefetch(pos->next); 1; }); \
485          pos = pos->next)
```

```
486
487 #define hlist_for_each_safe(pos, n, head) \
488     for (pos = (head)->first; pos && ({ n = pos->next; 1; }); \
489          pos = n)
490
498 #define hlist_for_each_entry(tpos, pos, head, member)            \
499     for (pos = (head)->first;                                    \
500          pos && ({ prefetch(pos->next); 1;}) &&                  \
501         ({ tpos = hlist_entry(pos, typeof(*tpos), member); 1;}); \
502          pos = pos->next)
503
510 #define hlist_for_each_entry_continue(tpos, pos, member)         \
511     for (pos = (pos)->next;                                      \
512          pos && ({ prefetch(pos->next); 1;}) &&                  \
513         ({ tpos = hlist_entry(pos, typeof(*tpos), member); 1;}); \
514          pos = pos->next)
515
522 #define hlist_for_each_entry_from(tpos, pos, member)             \
523     for (; pos && ({ prefetch(pos->next); 1;}) &&                \
524         ({ tpos = hlist_entry(pos, typeof(*tpos), member); 1;}); \
525          pos = pos->next)
526
535 #define hlist_for_each_entry_safe(tpos, pos, n, head, member)        \
536     for (pos = (head)->first;                                        \
537          pos && ({ n = pos->next; 1; }) &&                           \
538         ({ tpos = hlist_entry(pos, typeof(*tpos), member); 1;}); \
539          pos = n)
540
541
542 #endif
```

4.3 ums_lib.c File Reference

This file contains the implementation of all the functions of the library.

```
#include "ums_lib.h"
```

Functions

- int init_ums (void)

  Initialize/enable UMS in the program/process.
- int exit_ums (void)

  Exit/disable UMS in the program/process.
- int create_completion_list (void)

  Create completion list.
- int create_worker_thread (void(∗function)(void ∗), void ∗args, unsigned long stack_size)

  Create worker thread.
- int add_worker_thread (unsigned int completion_list_id, unsigned int worker_thread_id)

  Add worker thread to completion list.
- int enter_ums_scheduling_mode (void(∗function)(void ∗), unsigned long completion_list_id)

  Enter UMS scheduling mode.
- void ∗ convert_to_ums_thread (void ∗ums_thread_id)

  Convert pthread into ums thread(scheduler)
- int exit_ums_scheduling_mode (void)

  Exit UMS scheduling mode.
- int dequeue_completion_list_items (int ∗ready_wt_list)

  Dequeue completion list.
- int execute_worker_thread (int ∗ready_wt_list, int size, unsigned int worker_thread_id)

  Execute specific worker thread.
- int worker_thread_yield (yield_reason_t yield_reason)

Pause or finish worker thread.

- int get_next_ready_item (int *ready_wt_list, int size)

  Get next item from the list of ready worker threads.
- int check_ready_wt_list (int *ready_wt_list, int size)

  Check if the list of ready worker threads is empty.
- int open_dev (void)

  Open /dev/umsdevice device.
- int close_dev (void)

  Close /dev/umsdevice device.
- int get_wt_count_in_current_umst_cl (void)

  Get the number of worker threads in completion list of current ums thread(scheduler)
- completion_list_t * get_cl_with_id (unsigned int completion_list_id)

  Get completion list from cl_list_t.
- worker_thread_t * get_wt_with_id (unsigned int worker_thread_id)

  Get worker thread from worker_thread_list_t.
- ums_thread_t * get_umst_run_by_pthread (pthread_t current_pt)

  Get ums thread(scheduler) from ums_thread_list_t.
- int free_ums_thread_list (void)

  Clean the list of ums threads(schedulers)
- int free_cl_list (void)

  Clean the list of completion lists.
- int free_worker_thread_list (void)

  Clean the list of worker threads.
- void clean_memory ()

  Clean memory.
- **__attribute__** ((constructor))
- **__attribute__** ((destructor))

Variables

- int **fd** = -1
- cl_list_t cl_list
- worker_thread_list_t worker_thread_list
- ums_thread_list_t ums_thread_list

4.3.1 Detailed Description

This file contains the implementation of all the functions of the library.

Author

Sultan Umarbaev name.sul27@gmail.com

4.3.2 Function Documentation

4.3.2.1 add_worker_thread()

```
int add_worker_thread (
            unsigned int completion_list_id,
            unsigned int worker_thread_id )
```

Add worker thread to completion list.

Parameters

completion_list_id	the id of completion list to which worker thread is added
worker_thread_id	the id of worker thread that is added

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.3.2.2 check_ready_wt_list()

```
int check_ready_wt_list (
            int * ready_wt_list,
            int size )
```

Check if the list of ready worker threads is empty.

Parameters

ready_wt_list	the pointer to an array of ready to run worker thread ids
size	the size of the array

Returns

`int` 0 if false, otherwise true

4.3.2.3 clean_memory()

```
void clean_memory (
            void  )
```

Clean memory.

Clean up all created by the library data structures from the memory.

4.3.2.4 close_dev()

```
int close_dev (
            void  )
```

Close /dev/umsdevice device.

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.3.2.5 convert_to_ums_thread()

```
void * convert_to_ums_thread (
            void * ums_thread_id )
```

Convert pthread into ums thread(scheduler)

Convert current thread into ums thread(scheduler). This function is passed to pthread_create(), therefore the created pthread is converted into ums thread(scheduler).

Parameters

c int the id of ums thread(scheduler) into which to convert

4.3.2.6 create_completion_list()

```
int create_completion_list (
            void  )
```

Create completion list.

Create a new completion list and return a corresponding id. Add the completion list to cl_list.

Returns

`int` completion list id

4.3.2.7 create_worker_thread()

```
int create_worker_thread (
            void(*)(void *) function,
            void * args,
            unsigned long stack_size )
```

Create worker thread.

Create a new worker thread and return a corresponding id.

Parameters

function	the address of the starting function of the worker thread
args	the address of arguments allocated by the user passed to the function (first parameter)
stack_size	the stack size that is used for calculating the stack address after memory allocation with malloc function

Returns

`int` worker thread id

4.3.2.8 dequeue_completion_list_items()

```
int dequeue_completion_list_items (
            int * ready_wt_list )
```

Dequeue completion list.

Obtain a set of currently available worker threads to be run.

Parameters

ready_wt_list	the pointer to an allocated array of integers which will be filled with ready to run worker thread ids

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.3.2.9 enter_ums_scheduling_mode()

```
int enter_ums_scheduling_mode (
            void(*)(void *) function,
            unsigned long completion_list_id )
```

Enter UMS scheduling mode.

Create ums thread(scheduler) and pthread which will be converted to ums thread created earlier.

Parameters

function	an entry point function for the ums thread(scheduler), scheduling function
completion_list_id	the id of completion list with worker threads associated with ums thread(scheduler)

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.3.2.10 execute_worker_thread()

```
int execute_worker_thread (
            int * ready_wt_list,
            int size,
            unsigned int worker_thread_id )
```

Execute specific worker thread.

Execute worker thread given the id. The return result from ioctl call defines if requested worker thread is currently busy and handled by another ums thread(scheduler) or worker thread was already finished before. If it is busy then scheduler will try to execute next available thread from ready_wt_list. However, if it was already finished before ready_wt_list will be updated.

Parameters

ready_wt_list	the pointer to an array of ready to run worker thread ids
size	the size of the array
worker_thread_id	the id of the worker thread to be executed

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.3.2.11 exit_ums()

```
int exit_ums (
            void  )
```

Exit/disable UMS in the program/process.

Synchronize the execution of the ums threads(schedulers).

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.3.2.12 exit_ums_scheduling_mode()

```
int exit_ums_scheduling_mode (
            void  )
```

Exit UMS scheduling mode.

Convert from ums thread(scheduler) back to pthread.

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.3.2.13 free_cl_list()

```
int free_cl_list (
            void  )
```

Clean the list of completion lists.

Delete and free each item in the list of completion lists

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.3.2.14 free_ums_thread_list()

```
int free_ums_thread_list (
            void  )
```

Clean the list of ums threads(schedulers)

Delete and free each item in the list of ums threads(schedulers)

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.3.2.15 free_worker_thread_list()

```
int free_worker_thread_list (
            void  )
```

Clean the list of worker threads.

Delete and free each item in the list of worker threads

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.3.2.16 get_cl_with_id()

```
completion_list_t * get_cl_with_id (
            unsigned int completion_list_id )
```

Get completion list from cl_list_t.

Parameters

completion_list↩ _id	id of the completion list requested to be retrieved

Returns

`completion_list_t` the pointer to completion list with specific id

4.3.2.17 get_next_ready_item()

```
int get_next_ready_item (
            int * ready_wt_list,
            int size )
```

Get next item from the list of ready worker threads.

Parameters

ready_wt_list	the pointer to an array of ready to run worker thread ids
size	the size of the array

Returns

`int` worker thread id

4.3.2.18 get_umst_run_by_pthread()

```
ums_thread_t * get_umst_run_by_pthread (
            pthread_t current_pt )
```

Get ums thread(scheduler) from ums_thread_list_t.

Parameters

ums_thread↩ _id	id of the ums thread(scheduler) requested to be retrieved

Returns

`ums_thread_t` the pointer to ums thread(scheduler) with specific id

4.3.2.19 get_wt_count_in_current_umst_cl()

```
int get_wt_count_in_current_umst_cl (
            void  )
```

Get the number of worker threads in completion list of current ums thread(scheduler)

Returns

`int` number of worker threads

4.3.2.20 get_wt_with_id()

```
worker_thread_t * get_wt_with_id (
            unsigned int worker_thread_id )
```

Get worker thread from worker_thread_list_t.

Parameters

worker_thread_id	id of the worker thread requested to be retrieved

Returns

`worker_thread_t` the pointer to worker thread with specific id

4.3.2.21 init_ums()

```
int init_ums (
            void  )
```

Initialize/enable UMS in the program/process.

In order to start utilizing UMS mechanism, we need to enable UMS for the program/process.

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.3.2.22 open_dev()

```
int open_dev (
            void  )
```

Open /dev/umsdevice device.

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.3.2.23 worker_thread_yield()

```
int worker_thread_yield (
            yield_reason_t yield_reason )
```

Pause or finish worker thread.

Pause or finish worker thread deppending on the passed reason.

Parameters

yield_reason	reason which defines if worker thread should be paused or finished, yield_reason_t

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.3.3 Variable Documentation

4.3.3.1 cl_list

```
cl_list_t cl_list
```

Initial value:

```
= {
    .list = LIST_HEAD_INIT(cl_list.list),
    .cl_count = 0
}
```

4.3.3.2 ums_thread_list

```
ums_thread_list_t ums_thread_list
```

Initial value:

```
= {
    .list = LIST_HEAD_INIT(ums_thread_list.list),
    .ums_thread_count = 0
}
```

4.3.3.3 worker_thread_list

```
worker_thread_list_t worker_thread_list
```

Initial value:

```
= {
    .list = LIST_HEAD_INIT(worker_thread_list.list),
    .worker_thread_count = 0
}
```

4.4 ums_lib.h File Reference

This file is a header of the library.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <errno.h>
#include <fcntl.h>
#include <pthread.h>
#include "list.h"
#include "../module/device_shared.h"
```

Data Structures

- struct cl_list

  The list of completion lists.
- struct worker_thread_list

  The list of worker threads.
- struct ums_thread_list

  The list of ums threads(schedulers)
- struct completion_list

  The completion list of worker threads.
- struct worker_thread

  The worker thread.
- struct ums_thread

  The ums thread(scheduler)

Macros

- #define **UMS_DEVICE_PATH** "/dev/umsdevice"
- #define **UMS_LIB_LOG** "UMS lib: "
- #define **MIN_STACK_SIZE** 4096

Typedefs

- typedef struct cl_list cl_list_t

  The list of completion lists.
- typedef struct worker_thread_list worker_thread_list_t

  The list of worker threads.
- typedef struct ums_thread_list ums_thread_list_t

  The list of ums threads(schedulers)
- typedef struct completion_list completion_list_t

  The completion list of worker threads.
- typedef struct worker_thread worker_thread_t

  The worker thread.
- typedef struct ums_thread ums_thread_t

  The ums thread(scheduler)

Functions

- int init_ums (void)

  Initialize/enable UMS in the program/process.
- int exit_ums (void)

  Exit/disable UMS in the program/process.
- int create_completion_list (void)

  Create completion list.
- int create_worker_thread (void(∗function)(void ∗), void ∗args, unsigned long stack_size)

  Create worker thread.
- int add_worker_thread (unsigned int completion_list_id, unsigned int worker_thread_id)

  Add worker thread to completion list.
- int enter_ums_scheduling_mode (void(∗function)(void ∗), unsigned long completion_list_id)

  Enter UMS scheduling mode.
- void ∗ convert_to_ums_thread (void ∗ums_thread_id)

  Convert pthread into ums thread(scheduler)
- int exit_ums_scheduling_mode (void)

  Exit UMS scheduling mode.
- int dequeue_completion_list_items (int ∗ready_wt_list)

  Dequeue completion list.
- int execute_worker_thread (int ∗ready_wt_list, int size, unsigned int worker_thread_id)

  Execute specific worker thread.
- int worker_thread_yield (yield_reason_t yield_reason)

  Pause or finish worker thread.
- int get_next_ready_item (int ∗ready_wt_list, int size)

  Get next item from the list of ready worker threads.
- int check_ready_wt_list (int ∗ready_wt_list, int size)

  Check if the list of ready worker threads is empty.
- int open_dev (void)

Open /dev/umsdevice device.

- int close_dev (void)

  Close /dev/umsdevice device.
- int get_wt_count_in_current_umst_cl (void)

  Get the number of worker threads in completion list of current ums thread(scheduler)
- completion_list_t * get_cl_with_id (unsigned int completion_list_id)

  Get completion list from cl_list_t.
- worker_thread_t * get_wt_with_id (unsigned int worker_thread_id)

  Get worker thread from worker_thread_list_t.
- ums_thread_t * get_umst_run_by_pthread (pthread_t current_pt)

  Get ums thread(scheduler) from ums_thread_list_t.
- int free_ums_thread_list (void)

  Clean the list of ums threads(schedulers)
- int free_cl_list (void)

  Clean the list of completion lists.
- int free_worker_thread_list (void)

  Clean the list of worker threads.
- void clean_memory (void)

  Clean memory.
- **__attribute__** ((constructor)) void const ructor(void)
- **__attribute__** ((destructor)) void destructor(void)

4.4.1 Detailed Description

This file is a header of the library.

Copyright (C) 2021 Sultan Umarbaev name.sul27@gmail.com

This file is part of UMS implementation (Library).

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

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

You should have received a copy of the GNU General Public License along with UMS implementation (Library). If not, see http://www.gnu.org/licenses/.

This file contains all the data structures and function declarations of library

Author

Sultan Umarbaev name.sul27@gmail.com

4.4.2 Typedef Documentation

4.4.2.1 cl_list_t

```
typedef struct cl_list cl_list_t
```

The list of completion lists.

The purpose of this list is to store all completion lists created by the program

4.4.2.2 completion_list_t

```
typedef struct completion_list completion_list_t
```

The completion list of worker threads.

This is a node in the cl_list.

4.4.2.3 ums_thread_list_t

```
typedef struct ums_thread_list ums_thread_list_t
```

The list of ums threads(schedulers)

The purpose of this list is to store all ums threads(schedulers) created by the program

4.4.2.4 ums_thread_t

```
typedef struct ums_thread ums_thread_t
```

The ums thread(scheduler)

This is a node in the ums_thread_list.

4.4.2.5 worker_thread_list_t

```
typedef struct worker_thread_list worker_thread_list_t
```

The list of worker threads.

The purpose of this list is to store all worker threads created by the program

4.4.2.6 worker_thread_t

```
typedef struct worker_thread worker_thread_t
```

The worker thread.

This is a node in the worker_thread_list.

4.4.3 Function Documentation

4.4.3.1 add_worker_thread()

```
int add_worker_thread (
            unsigned int completion_list_id,
            unsigned int worker_thread_id )
```

Add worker thread to completion list.

Parameters

completion_list↩ _id	the id of completion list to which worker thread is added
worker_thread_id	the id of worker thread that is added

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.4.3.2 check_ready_wt_list()

```
int check_ready_wt_list (
            int * ready_wt_list,
            int size )
```

Check if the list of ready worker threads is empty.

Parameters

ready_wt_list	the pointer to an array of ready to run worker thread ids
size	the size of the array

Returns

`int` 0 if false, otherwise true

4.4.3.3 clean_memory()

```
void clean_memory (
            void  )
```

Clean memory.

Clean up all created by the library data structures from the memory.

4.4.3.4 close_dev()

```
int close_dev (
            void  )
```

Close /dev/umsdevice device.

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.4.3.5 convert_to_ums_thread()

```
void * convert_to_ums_thread (
            void * ums_thread_id )
```

Convert pthread into ums thread(scheduler)

Convert current thread into ums thread(scheduler). This function is passed to pthread_create(), therefore the created pthread is converted into ums thread(scheduler).

Parameters

c int the id of ums thread(scheduler) into which to convert

4.4.3.6 create_completion_list()

```
int create_completion_list (
            void  )
```

Create completion list.

Create a new completion list and return a corresponding id. Add the completion list to cl_list.

Returns

`int` completion list id

4.4.3.7 create_worker_thread()

```
int create_worker_thread (
            void(*)(void *) function,
            void * args,
            unsigned long stack_size )
```

Create worker thread.

Create a new worker thread and return a corresponding id.

Parameters

function	the address of the starting function of the worker thread
args	the address of arguments allocated by the user passed to the function (first parameter)
stack_size	the stack size that is used for calculating the stack address after memory allocation with malloc function

Returns

`int` worker thread id

4.4.3.8 dequeue_completion_list_items()

```
int dequeue_completion_list_items (
            int * ready_wt_list )
```

Dequeue completion list.

Obtain a set of currently available worker threads to be run.

Parameters

ready_wt_list	the pointer to an allocated array of integers which will be filled with ready to run worker thread ids

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.4.3.9 enter_ums_scheduling_mode()

```
int enter_ums_scheduling_mode (
            void(*)(void *) function,
            unsigned long completion_list_id )
```

Enter UMS scheduling mode.

Create ums thread(scheduler) and pthread which will be converted to ums thread created earlier.

Parameters

function	an entry point function for the ums thread(scheduler), scheduling function
completion_list_id	the id of completion list with worker threads associated with ums thread(scheduler)

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.4.3.10 execute_worker_thread()

```
int execute_worker_thread (
            int * ready_wt_list,
```

```
            int size,
            unsigned int worker_thread_id )
```

Execute specific worker thread.

Execute worker thread given the id. The return result from ioctl call defines if requested worker thread is currently busy and handled by another ums thread(scheduler) or worker thread was already finished before. If it is busy then scheduler will try to execute next available thread from ready_wt_list. However, if it was already finished before ready_wt_list will be updated.

Parameters

ready_wt_list	the pointer to an array of ready to run worker thread ids
size	the size of the array
worker_thread_id	the id of the worker thread to be executed

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.4.3.11 exit_ums()

```
int exit_ums (
            void  )
```

Exit/disable UMS in the program/process.

Synchronize the execution of the ums threads(schedulers).

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.4.3.12 exit_ums_scheduling_mode()

```
int exit_ums_scheduling_mode (
            void  )
```

Exit UMS scheduling mode.

Convert from ums thread(scheduler) back to pthread.

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.4.3.13 free_cl_list()

```
int free_cl_list (
            void  )
```

Clean the list of completion lists.

Delete and free each item in the list of completion lists

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.4.3.14 free_ums_thread_list()

```
int free_ums_thread_list (
            void  )
```

Clean the list of ums threads(schedulers)

Delete and free each item in the list of ums threads(schedulers)

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.4.3.15 free_worker_thread_list()

```
int free_worker_thread_list (
            void  )
```

Clean the list of worker threads.

Delete and free each item in the list of worker threads

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.4.3.16 get_cl_with_id()

```
completion_list_t * get_cl_with_id (
            unsigned int completion_list_id )
```

Get completion list from cl_list_t.

Parameters

completion_list↩ _id	id of the completion list requested to be retrieved

Returns

`completion_list_t` the pointer to completion list with specific id

4.4.3.17 get_next_ready_item()

```
int get_next_ready_item (
            int * ready_wt_list,
            int size )
```

Get next item from the list of ready worker threads.

Parameters

ready_wt_list	the pointer to an array of ready to run worker thread ids
size	the size of the array

Returns

`int` worker thread id

4.4.3.18 get_umst_run_by_pthread()

```
ums_thread_t * get_umst_run_by_pthread (
            pthread_t current_pt )
```

Get ums thread(scheduler) from ums_thread_list_t.

Parameters

ums_thread↩ _id	id of the ums thread(scheduler) requested to be retrieved

Returns

`ums_thread_t` the pointer to ums thread(scheduler) with specific id

4.4.3.19 get_wt_count_in_current_umst_cl()

```
int get_wt_count_in_current_umst_cl (
            void  )
```

Get the number of worker threads in completion list of current ums thread(scheduler)

Returns

`int` number of worker threads

4.4.3.20 get_wt_with_id()

```
worker_thread_t * get_wt_with_id (
            unsigned int worker_thread_id )
```

Get worker thread from worker_thread_list_t.

Parameters

worker_thread_id	id of the worker thread requested to be retrieved

Returns

`worker_thread_t` the pointer to worker thread with specific id

4.4.3.21 init_ums()

```
int init_ums (
            void  )
```

Initialize/enable UMS in the program/process.

In order to start utilizing UMS mechanism, we need to enable UMS for the program/process.

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.4.3.22 open_dev()

```
int open_dev (
            void  )
```

Open /dev/umsdevice device.

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.4.3.23 worker_thread_yield()

```
int worker_thread_yield (
            yield_reason_t yield_reason )
```

Pause or finish worker thread.

Pause or finish worker thread deppending on the passed reason.

Parameters

yield_reason	reason which defines if worker thread should be paused or finished, yield_reason_t

Returns

`int` exit code 0 for success, otherwise a corresponding error code

4.5 ums_lib.h

Go to the documentation of this file.

```
1
30 #pragma once
31
32 #include <stdio.h>
33 #include <stdlib.h>
34 #include <unistd.h>
35 #include <sys/ioctl.h>
36 #include <errno.h>
37 #include <fcntl.h>
38 #include <pthread.h>
39
40 #include "list.h"
41 #include "../module/device_shared.h"
42
43 #define UMS_DEVICE_PATH "/dev/umsdevice"
44 #define UMS_LIB_LOG "UMS lib: "
45
46 #define MIN_STACK_SIZE 4096
47
48 /*
49  * Structs
50  */
51
58 typedef struct cl_list {
59     struct list_head list;
```

```
60     unsigned int cl_count;
61 } cl_list_t;
62
69 typedef struct worker_thread_list {
70     struct list_head list;
71     unsigned int worker_thread_count;
72 } worker_thread_list_t;
73
80 typedef struct ums_thread_list {
81     struct list_head list;
82     unsigned int ums_thread_count;
83 } ums_thread_list_t;
84
91 typedef struct completion_list {
92     unsigned int id;
93     unsigned int worker_thread_count;
94     struct list_head list;
95 } completion_list_t;
96
103 typedef struct worker_thread {
104     unsigned int id;
105     worker_thread_params_t *params;
106     struct list_head list;
107 } worker_thread_t;
108
115 typedef struct ums_thread {
116     unsigned int id;
117     pthread_t pt;
118     ums_thread_params_t *params;
119     struct list_head list;
120 } ums_thread_t;
121
122 /*
123  * Functions
124  */
125 int init_ums(void);
126 int exit_ums(void);
127 int create_completion_list(void);
128 int create_worker_thread(void (*function)(void *), void *args, unsigned long stack_size);
129 int add_worker_thread(unsigned int completion_list_id, unsigned int worker_thread_id);
130 int enter_ums_scheduling_mode(void (*function)(void *), unsigned long completion_list_id);
131 void *convert_to_ums_thread(void *ums_thread_id);
132 int exit_ums_scheduling_mode(void);
133 int dequeue_completion_list_items(int *ready_wt_list);
134 int execute_worker_thread(int *ready_wt_list, int size, unsigned int worker_thread_id);
135 int worker_thread_yield(yield_reason_t yield_reason);
136
137 int get_next_ready_item(int *ready_wt_list, int size);
138 int check_ready_wt_list(int *ready_wt_list, int size);
139
140 /*
141  * Auxiliary functions
142  */
143 int open_dev(void);
144 int close_dev(void);
145 int get_wt_count_in_current_umst_cl(void);
146 completion_list_t *get_cl_with_id(unsigned int completion_list_id);
147 worker_thread_t *get_wt_with_id(unsigned int worker_thread_id);
148 ums_thread_t *get_umst_run_by_pthread(pthread_t current_pt);
149 int free_ums_thread_list(void);
150 int free_cl_list(void);
151 int free_worker_thread_list(void);
152 void clean_memory(void);
153
154 __attribute__((constructor)) void constructor(void);
155 __attribute__((destructor)) void destructor(void);
```

Index