UMS: Library

Generated by Doxygen 1.9.2

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 hlist_head Struct Reference	5
3.2 hlist_node Struct Reference	5
3.3 list_head Struct Reference	5
3.3.1 Detailed Description	6
3.4 list_params Struct Reference	6
3.4.1 Detailed Description	6
3.4.2 Field Documentation	6
3.4.2.1 size	6
3.4.2.2 state	6
3.4.2.3 worker_count	7
3.4.2.4 workers	7
3.5 scheduler_params Struct Reference	7
3.5.1 Detailed Description	7
3.5.2 Field Documentation	
3.5.2.1 clid	7
3.5.2.2 core_id	8
3.5.2.3 entry_point	
3.5.2.4 sid	
3.6 ums_completion_list Struct Reference	8
3.6.1 Detailed Description	
3.6.2 Field Documentation	
3.6.2.1 count	9
3.7 ums_completion_list_node Struct Reference	
3.7.1 Detailed Description	
3.7.2 Field Documentation	
3.7.2.1 clid	
3.7.2.2 list_params	
3.7.2.3 state	
3.7.2.4 worker_count	_
3.8 ums_scheduler Struct Reference	_
3.8.1 Detailed Description	_
3.8.2 Field Documentation	
3.8.2.1 list_params	
3.8.2.2 sched_params	_
3.8.2.3 tid	
3.8.2.4 wid	
O.O.E.T WILL	1.1

	3.9 ums_scheduler_list Struct Reference	11
	3.9.1 Detailed Description	11
	3.9.2 Field Documentation	11
	3.9.2.1 count	12
	3.10 ums_worker Struct Reference	12
	3.10.1 Detailed Description	12
	3.10.2 Field Documentation	12
	3.10.2.1 state	12
	3.10.2.2 wid	12
	3.10.2.3 worker_params	13
	3.11 ums_worker_list Struct Reference	13
	3.11.1 Detailed Description	13
	3.11.2 Field Documentation	13
	3.11.2.1 count	13
	3.12 worker_params Struct Reference	13
	3.12.1 Detailed Description	14
	3.12.2 Field Documentation	14
	3.12.2.1 clid	14
	3.12.2.2 entry_point	14
	3.12.2.3 function_args	14
	3.12.2.4 stack_addr	14
	3.12.2.5 stack_size	14
4 1	Tile Decumentation	15
4 1	File Documentation 4.1 const.h File Reference	
	4.1 Const.n File Reference	15
	•	17
	4.1.2 Enumeration Type Documentation	17
	4.1.2.1 state	
	4.1.2.2 worker_status	18
	4.2 const.h	18
	4.3 list.h File Reference	19
	4.3.1 Detailed Description	20
	4.3.2 Macro Definition Documentation	21 21
	4.3.2.1list_for_each	
	4.3.2.2 container_of	21
	4.3.2.3 hlist_for_each	21
	4.3.2.4 hlist_for_each_entry	22
	4.3.2.5 hlist_for_each_entry_continue	22
	4.3.2.6 hlist_for_each_entry_from	22
	4.3.2.7 hlist_for_each_entry_safe	23
	4.3.2.8 hlist_for_each_safe	23
	4.3.2.9 INIT_LIST_HEAD	23

4.3.2.10 list_entry	23
4.3.2.11 list_for_each	24
4.3.2.12 list_for_each_entry	24
4.3.2.13 list_for_each_entry_continue	24
4.3.2.14 list_for_each_entry_reverse	24
4.3.2.15 list_for_each_entry_safe	25
4.3.2.16 list_for_each_entry_safe_continue	25
4.3.2.17 list_for_each_entry_safe_reverse	25
4.3.2.18 list_for_each_prev	26
4.3.2.19 list_for_each_safe	26
4.3.2.20 list_prepare_entry	26
4.3.2.21 offsetof	26
4.4 list.h	27
4.5 ums_lib.c File Reference	30
4.5.1 Detailed Description	32
4.5.2 Function Documentation	33
4.5.2.1 check_if_completion_list_exists()	33
4.5.2.2 check_if_scheduler_exists()	34
4.5.2.3 check_if_worker_exists()	34
4.5.2.4 cleanup()	34
4.5.2.5 close_device()	35
4.5.2.6 open_device()	35
4.5.2.7 ums_create_completion_list()	35
4.5.2.8 ums_create_scheduler()	35
4.5.2.9 ums_create_worker_thread()	36
4.5.2.10 ums_dequeue_completion_list_items()	36
4.5.2.11 ums_enter()	37
4.5.2.12 ums_enter_scheduling_mode()	37
4.5.2.13 ums_execute_thread()	37
4.5.2.14 ums_exit()	38
4.5.2.15 ums_exit_scheduling_mode()	38
4.5.2.16 ums_get_next_worker_thread()	38
4.5.2.17 ums_thread_exit()	38
4.5.2.18 ums_thread_pause()	39
4.5.2.19 ums_thread_yield()	39
4.5.3 Variable Documentation	39
4.5.3.1 completion_lists	39
4.5.3.2 schedulers	40
4.5.3.3 workers	40
4.6 ums_lib.h File Reference	40
4.6.1 Detailed Description	42
4.6.2 Function Documentation	43

	4.6.2.1 check_if_completion_list_exists()	43
	4.6.2.2 check_if_scheduler_exists()	43
	4.6.2.3 check_if_worker_exists()	43
	4.6.2.4 cleanup()	44
	4.6.2.5 close_device()	44
	4.6.2.6 open_device()	44
	4.6.2.7 ums_create_completion_list()	44
	4.6.2.8 ums_create_worker_thread()	44
	4.6.2.9 ums_dequeue_completion_list_items()	45
	4.6.2.10 ums_enter()	45
	4.6.2.11 ums_enter_scheduling_mode()	45
	4.6.2.12 ums_execute_thread()	46
	4.6.2.13 ums_exit()	46
	4.6.2.14 ums_exit_scheduling_mode()	46
	4.6.2.15 ums_get_next_worker_thread()	47
	4.6.2.16 ums_thread_exit()	47
	4.6.2.17 ums_thread_pause()	47
4.7 ums_lib.h		48
Index		49

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

hlist_head	5
hlist_node	5
list_head	5
list_params	
Parameters that are created by the scheduler and passed to dequeue the completion list items	6
scheduler_params	
Parameters that are passed in order to create a scheduler	7
ums_completion_list	
The list of the completion lists created by the process	8
ums_completion_list_node	
Represents a node in the ums_completion_list	9
ums_scheduler	
Represents a node in the ums_scheduler_list	10
ums_scheduler_list	
The list of the schedulers created by the process	11
ums_worker	
Represents a node in the ums_worker_list	12
ums_worker_list	
The list of the worker threads created by the process	13
worker_params	
Parameters that are passed in order to create a worker thread	13

2 Data Structure Index

Chapter 2

File Index

2.1 File List

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

const.h		
	Set of data structures and other constant variables used by UMS library	15
list.h		
	Implementation of the Linux kernel linked list and hash list data structures for user space	19
ums_lib.	c	
	Contains implementations of the essential UMS library functions	30
ums_lib.	h	
	The header that contains essential UMS library functions and has to be included by the user in	
	order to use the UMS library	40

File Index

Chapter 3

Data Structure Documentation

3.1 hlist_head Struct Reference

Data Fields

struct hlist_node * first

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

• list.h

3.2 hlist_node Struct Reference

Data Fields

```
struct hlist_node * next
```

struct hlist_node ** pprev

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

· list.h

3.3 list_head Struct Reference

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

Data Fields

```
struct list_head * next
```

struct list_head * prev

3.3.1 Detailed Description

Simple doubly linked list implementation.

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

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

· list.h

3.4 list_params Struct Reference

Parameters that are created by the scheduler and passed to dequeue the completion list items

```
#include <const.h>
```

Data Fields

- · unsigned int size
- unsigned int worker_count
- · state_t state
- ums_wid_t workers []

3.4.1 Detailed Description

Parameters that are created by the scheduler and passed to dequeue the completion list items

3.4.2 Field Documentation

3.4.2.1 size

```
unsigned int list_params::size
```

Size of the worker thread array

3.4.2.2 state

```
state_t list_params::state
```

Tracks the state of the completion list which is set by the kernel module after a dequeue call

3.4.2.3 worker_count

```
unsigned int list_params::worker_count
```

Tracks the quantity of the available workers and used as state indicator for scheduler to perform a new dequeue call

3.4.2.4 workers

```
ums_wid_t list_params::workers[]
```

Array of worker threads. Stores ID of worker threads in case they are available to be scheduled (when worker thread is finished, scheduler replaces ID with -1 value)

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

· const.h

3.5 scheduler_params Struct Reference

Parameters that are passed in order to create a scheduler

```
#include <const.h>
```

Data Fields

- unsigned long entry_point
- ums_clid_t clid
- ums_sid_t sid
- int core id

3.5.1 Detailed Description

Parameters that are passed in order to create a scheduler

3.5.2 Field Documentation

3.5.2.1 clid

```
ums_clid_t scheduler_params::clid
```

ID of the completion list that is assigned to the scheduler

3.5.2.2 core_id

```
int scheduler_params::core_id
```

ID of the CPU core that is assigned to the scheduler (It is handled automatically by the library, no user input required)

3.5.2.3 entry_point

```
unsigned long scheduler_params::entry_point
```

Function pointer and an entry point set by a user, that serves as a starting point of the scheduler. It is a scheduling function that determines the next thread to be scheduled

3.5.2.4 sid

```
ums_sid_t scheduler_params::sid
```

ID of the scheduler which is set by the kernel module

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

· const.h

3.6 ums completion list Struct Reference

The list of the completion lists created by the process

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

Data Fields

- struct list_head list
- unsigned int count

3.6.1 Detailed Description

The list of the completion lists created by the process

3.6.2 Field Documentation

3.6.2.1 count

```
unsigned int ums_completion_list::count
```

Number of completion lists created

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

• ums_lib.h

3.7 ums completion list node Struct Reference

Represents a node in the ums_completion_list

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

Data Fields

- ums_clid_t clid
- state_t state
- unsigned int worker_count
- struct list_head list
- list_params_t * list_params

3.7.1 Detailed Description

Represents a node in the ums_completion_list

3.7.2 Field Documentation

3.7.2.1 clid

```
ums_clid_t ums_completion_list_node::clid
```

Completion list ID

3.7.2.2 list_params

```
{\tt list\_params\_t*} \ {\tt ums\_completion\_list\_node::list\_params
```

Parameters that are created by the scheduler and passed to dequeue the completion list items list_params

3.7.2.3 state

```
state_t ums_completion_list_node::state
```

State of the completion list

3.7.2.4 worker_count

```
unsigned int ums_completion_list_node::worker_count
```

Number of worker threads assigned to the completion list

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

• ums_lib.h

3.8 ums_scheduler Struct Reference

Represents a node in the ums_scheduler_list

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

Data Fields

- struct list_head list
- pthread_t tid
- ums_wid_t wid
- scheduler_params_t * sched_params
- list_params_t * list_params

3.8.1 Detailed Description

Represents a node in the ums_scheduler_list

3.8.2 Field Documentation

3.8.2.1 list_params

```
{\tt list\_params\_t*} \ {\tt ums\_scheduler::list\_params}
```

Parameters that are created by the scheduler and passed to dequeue the completion list items list_params

3.8.2.2 sched_params

```
scheduler_params_t* ums_scheduler::sched_params
```

Parameters that are passed in order to create a scheduler scheduler_params

3.8.2.3 tid

```
pthread_t ums_scheduler::tid
```

Pthread ID

3.8.2.4 wid

```
ums_wid_t ums_scheduler::wid
```

Worker thread ID

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

• ums_lib.h

3.9 ums_scheduler_list Struct Reference

The list of the schedulers created by the process

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

Data Fields

- struct list_head list
- · unsigned int count

3.9.1 Detailed Description

The list of the schedulers created by the process

3.9.2 Field Documentation

3.9.2.1 count

```
unsigned int ums_scheduler_list::count
```

Number of scheduler created

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

• ums_lib.h

3.10 ums_worker Struct Reference

Represents a node in the ums_worker_list

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

Data Fields

- ums_wid_t wid
- state_t state
- struct list head list
- worker_params_t * worker_params

3.10.1 Detailed Description

Represents a node in the ums_worker_list

3.10.2 Field Documentation

3.10.2.1 state

```
state_t ums_worker::state
```

State of worker thread's progress

3.10.2.2 wid

```
ums_wid_t ums_worker::wid
```

Worker thread ID

3.10.2.3 worker_params

```
worker_params_t* ums_worker::worker_params
```

Parameters that are passed in order to create a worker thread worker_params

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

• ums_lib.h

3.11 ums_worker_list Struct Reference

The list of the worker threads created by the process

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

Data Fields

- struct list_head list
- · unsigned int count

3.11.1 Detailed Description

The list of the worker threads created by the process

3.11.2 Field Documentation

3.11.2.1 count

```
unsigned int ums_worker_list::count
```

Number of worker threads created

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

• ums_lib.h

3.12 worker_params Struct Reference

Parameters that are passed in order to create a worker thread

```
#include <const.h>
```

Data Fields

- unsigned long entry_point
- unsigned long function_args
- unsigned long stack_size
- unsigned long stack_addr
- · ums clid t clid

3.12.1 Detailed Description

Parameters that are passed in order to create a worker thread

3.12.2 Field Documentation

3.12.2.1 clid

```
ums_clid_t worker_params::clid
```

ID of the completion list where worker thread is assigned to

3.12.2.2 entry_point

```
unsigned long worker_params::entry_point
```

Function pointer and an entry point set by a user, that serves as a starting point of the worker thread

3.12.2.3 function_args

```
unsigned long worker_params::function_args
```

Pointer of the function arguments that are passed to the entry point/function

3.12.2.4 stack_addr

```
unsigned long worker_params::stack_addr
```

Address of the stack allocated by the UMS library

3.12.2.5 stack size

```
unsigned long worker_params::stack_size
```

Stack size of the worker thread set by a user

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

const.h

Chapter 4

File Documentation

4.1 const.h File Reference

Set of data structures and other constant variables used by UMS library.

#include <linux/ioctl.h>

Data Structures

· struct list params

Parameters that are created by the scheduler and passed to dequeue the completion list items

struct worker_params

Parameters that are passed in order to create a worker thread

struct scheduler_params

Parameters that are passed in order to create a scheduler

Macros

- #define UMS_NAME "ums"
- #define UMS DEVICE "/dev/ums"
- #define UMS_IOC_MAGIC 'R'
- #define UMS_ENTER IO(UMS_IOC_MAGIC, 1)
- #define UMS_EXIT_IO(UMS_IOC_MAGIC, 2)
- #define UMS_CREATE_LIST_IO(UMS_IOC_MAGIC, 3)
- #define UMS CREATE WORKER IOW(UMS IOC MAGIC, 4, unsigned long)
- #define UMS_ENTER_SCHEDULING_MODE_IOWR(UMS_IOC_MAGIC, 5, unsigned long)
- #define UMS_EXIT_SCHEDULING_MODE _IO(UMS_IOC_MAGIC, 6)
- #define **UMS_EXECUTE_THREAD** _IOW(UMS_IOC_MAGIC, 7, unsigned long)
- #define UMS_THREAD_YIELD _IOW(UMS_IOC_MAGIC, 8, unsigned long)
- #define UMS_DEQUEUE_COMPLETION_LIST_ITEMS _IOWR(UMS_IOC_MAGIC, 9, unsigned long)
- #define UMS_SUCCESS 0

Succesful execution.

• #define UMS_ERROR 1

Error.

#define UMS ERROR PROCESS NOT FOUND 1000

Process is not managed by UMS kernel module.

#define UMS ERROR PROCESS ALREADY EXISTS 1001

Process is already managed by UMS kernel module.

#define UMS_ERROR_COMPLETION_LIST_NOT_FOUND 1002

Completion list cannot be found.

#define UMS ERROR SCHEDULER NOT FOUND 1003

Scheduler cannot be found.

#define UMS_ERROR_WORKER_NOT_FOUND 1004

Worker thread cannot be found.

#define UMS ERROR STATE RUNNING 1005

The object is still running, thus cannot be modified, updated, deleted.

#define UMS_ERROR_CMD_IS_NOT_ISSUED_BY_MAIN_THREAD 1006

The command is not issued by the main process thread, e.g. ums_exit()

#define UMS_ERROR_WORKER_ALREADY_RUNNING

The worker thread is already running.

• #define UMS_ERROR_WRONG_INPUT 1008

Wrong input.

• #define UMS ERROR CMD IS NOT ISSUED BY SCHEDULER 1009

The command is not issued by the scheduler.

• #define UMS_ERROR_CMD_IS_NOT_ISSUED_BY_WORKER 1010

The command is not issued by the worker.

#define UMS_ERROR_WORKER_ALREADY_FINISHED 1011

The worker thread has already finished execution.

#define UMS_ERROR_NO_AVAILABLE_WORKERS 1012

No worker threads are available.

• #define UMS ERROR COMPLETION LIST ALREADY FINISHED 1013

All worker threads in the completion list have finished execution.

#define UMS_ERROR_FAILED_TO_CREATE_PROC_ENTRY 1014

Failed to create proc entry.

#define UMS ERROR FAILED TO PROC OPEN 1015

Failed to open proc entry.

#define UMS ERROR COMPLETION LIST IS USED AND CANNOT BE MODIFIED 1016

The completion list is being used, thus cannot be modified.

#define UMS_MIN_STACK_SIZE 4096

The minimum stack size of the worker thread

Typedefs

· typedef enum state state t

States of processes, completion lists and threads (schedulers, worker threads)

typedef enum worker_status worker_status_t

Status of the worker thread Used as a parameter that is passed for pausing or completing the worker thread.

· typedef unsigned int ums sid t

Scheduler ID

typedef unsigned int ums_wid_t

Worker thread ID

· typedef unsigned int ums clid t

Completion list ID

typedef struct list_params list_params_t

4.1 const.h File Reference 17

Parameters that are created by the scheduler and passed to dequeue the completion list items

typedef struct worker_params worker_params_t

Parameters that are passed in order to create a worker thread

• typedef struct scheduler_params scheduler_params_t

Parameters that are passed in order to create a scheduler

Enumerations

• enum state { IDLE , RUNNING , FINISHED }

States of processes, completion lists and threads (schedulers, worker threads)

enum worker_status { PAUSE , FINISH }

Status of the worker thread Used as a parameter that is passed for pausing or completing the worker thread.

4.1.1 Detailed Description

Set of data structures and other constant variables used by UMS library.

Copyright (C) 2021 Bektur Umarbaev hrafnulf13@gmail.com

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

UMS 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 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 library. If not, see http://www.gnu.org/licenses/.

Author

Bektur Umarbaev hrafnulf13@gmail.com

Date

4.1.2 Enumeration Type Documentation

4.1.2.1 state

enum state

States of processes, completion lists and threads (schedulers, worker threads)

Enumerator

IDLE	g ,
	searches for available worker threads to run; Completion list has available worker threads to be
	scheduled
RUNNING	Represents the state when worker thread is scheduled and ran by the scheduler; When
	scheduler handles worker thread; Completion list is currently used and can't be modified
FINISHED	Represents the state when worker thread has been completed; When scheduler has completed
	all scheduling work with a completion list; All completion list's worker threads has been completed

4.1.2.2 worker status

```
enum worker_status
```

Status of the worker thread Used as a parameter that is passed for pausing or completing the worker thread.

Enumerator

PAUSE	Used for pausing a worker thread: ums_thread_pause() == ums_thread_yield(PAUSE)
FINISH	Used for completing a worker thread: ums_thread_exit() == ums_thread_yield(FINISH)

4.2 const.h

Go to the documentation of this file.

```
29 #pragma once
30
31 #include <linux/ioctl.h>
32
34 * Definitions
35 */
36
37 #define UMS_NAME
38 #define UMS_DEVICE
                                                     "ums"
                                                     "/dev/ums"
39 #define UMS_IOC_MAGIC
41 /*
42 * IOCTL definitions 43 */
                                                                                _IO(UMS_IOC_MAGIC, 1)
44 #define UMS_ENTER
                                                                                _IO(UMS_IOC_MAGIC, 2)
45 #define UMS_EXIT
46 #define UMS_CREATE_LIST
                                                                                _IO(UMS_IOC_MAGIC, 3)
46 #define UMS_CREATE_LIST

47 #define UMS_CREATE_WORKER

48 #define UMS_ENTER_SCHEDULING_MODE

49 #define UMS_EXIT_SCHEDULING_MODE

50 #define UMS_EXECUTE_THREAD

51 #define UMS_THREAD_YIELD

52 #define UMS_DEQUEUE_COMPLETION_LIST_ITEMS

10 (UMS_IOC_MAGIC, 4, unsigned long)

10 (UMS_IOC_MAGIC, 5, unsigned long)

10 (UMS_IOC_MAGIC, 7, unsigned long)

10 (UMS_IOC_MAGIC, 8, unsigned long)

10 (UMS_IOC_MAGIC, 9, unsigned long)
54 /*
55 * Errors and return values
56 */
57 #define UMS_SUCCESS
58 #define UMS_ERROR
59 #define UMS_ERROR_PROCESS_NOT_FOUND
60 #define UMS_ERROR_PROCESS_ALREADY_EXISTS
```

4.3 list.h File Reference

```
61 #define UMS_ERROR_COMPLETION_LIST_NOT_FOUND
62 #define UMS_ERROR_SCHEDULER_NOT_FOUND
63 #define UMS ERROR WORKER NOT FOUND
                                                                             1004
64 #define UMS_ERROR_STATE_RUNNING
                                                                             1005
65 #define UMS_ERROR_CMD_IS_NOT_ISSUED_BY_MAIN_THREAD
                                                                             1006
66 #define UMS_ERROR_WORKER_ALREADY_RUNNING
67 #define UMS_ERROR_WRONG_INPUT
                                                                             1008
68 #define UMS_ERROR_CMD_IS_NOT_ISSUED_BY_SCHEDULER
                                                                             1009
69 #define UMS_ERROR_CMD_IS_NOT_ISSUED_BY_WORKER
70 #define UMS_ERROR_WORKER_ALREADY_FINISHED
71 #define UMS_ERROR_NO_AVAILABLE_WORKERS
72 #define UMS_ERROR_COMPLETION_LIST_ALREADY_FINISHED
73 #define UMS_ERROR_FAILED_TO_CREATE_PROC_ENTRY
                                                                             1014
74 #define UMS_ERROR_FAILED_TO_PROC_OPEN
                                                                             1015
75 #define UMS_ERROR_COMPLETION_LIST_IS_USED_AND_CANNOT_BE_MODIFIED
81 #define UMS_MIN_STACK_SIZE
87 typedef enum state {
      IDLE,
RUNNING,
88
89
       FINISHED
90
91 } state_t;
97 typedef enum worker_status {
98
      PAUSE.
      FINISH
99
100 } worker_status_t;
101
106 typedef unsigned int ums_sid_t;
107
112 typedef unsigned int ums_wid_t;
113
118 typedef unsigned int ums_clid_t;
119
124 typedef struct list_params {
125
       unsigned int size;
126
        unsigned int worker_count;
127
       state_t state;
       ums_wid_t workers[];
128
129 } list_params_t;
130
135 typedef struct worker_params {
136
      unsigned long entry_point;
137
       unsigned long function_args;
138
       unsigned long stack_size;
139
      unsigned long stack_addr;
140
       ums_clid_t clid;
141 } worker_params_t;
142
147 typedef struct scheduler_params {
148 unsigned long entry_point;
149
       ums_clid_t clid;
150
       ums_sid_t sid;
        int core_id;
152 } scheduler_params_t;
```

4.3 list.h File Reference

Implementation of the Linux kernel linked list and hash list data structures for user space.

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

Data Structures

- · struct list head
- · struct hlist head
- · struct hlist node

from other kernel headers

- #define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
- #define container of(ptr, type, member)
- #define LIST_POISON1 ((void *) 0x00100100)
- #define LIST_POISON2 ((void *) 0x00200200)
- #define LIST_HEAD_INIT(name) { &(name), &(name) }
- #define LIST_HEAD(name) struct list head name = LIST_HEAD_INIT(name)
- #define INIT_LIST_HEAD(ptr)
- #define list_entry(ptr, type, member) container_of(ptr, type, member)
- #define list for each(pos, head)
- #define __list_for_each(pos, head) for (pos = (head)->next; pos != (head); pos = pos->next)
- #define list for each prev(pos, head)
- #define list for each safe(pos, n, head)
- #define list for each entry(pos, head, member)
- #define list for each entry reverse(pos, head, member)
- #define list prepare entry(pos, head, member) ((pos) ?: list entry(head, typeof(*pos), member))
- #define list_for_each_entry_continue(pos, head, member)
- #define list_for_each_entry_safe(pos, n, head, member)
- #define list_for_each_entry_safe_continue(pos, n, head, member)
- #define list_for_each_entry_safe_reverse(pos, n, head, member)
- #define **HLIST_HEAD_INIT** { .first = NULL }
- #define HLIST_HEAD(name) struct hlist head name = { .first = NULL }
- #define INIT_HLIST_HEAD(ptr) ((ptr)->first = NULL)
- #define INIT_HLIST_NODE(ptr) ((ptr)->next = NULL, (ptr)->pprev = NULL)
- #define **hlist_entry**(ptr, type, member) **container_of**(ptr,type,member)
- #define hlist_for_each(pos, head)
- #define hlist_for_each_safe(pos, n, head)
- #define hlist_for_each_entry(tpos, pos, head, member)
- #define hlist_for_each_entry_continue(tpos, pos, member)
- #define hlist_for_each_entry_from(tpos, pos, member)
- #define hlist for each entry safe(tpos, pos, n, head, member)

4.3.1 Detailed Description

Implementation of the Linux kernel linked list and hash list data structures for user space.

Copyright (C) 2021 Bektur Umarbaev hrafnulf13@gmail.com

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

UMS 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 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 library. If not, see http←://www.gnu.org/licenses/.

The following file is shared under GPL license and was downloaded from http://www.mcs.anl.com/ov/~kazutomo/list/list.h The file without modifications was used for the implementation of UMS library. Here by copyright, credits are attributed to Kazutomo Yoshii kazutomo@mcs.anl.gov.

4.3 list.h File Reference 21

Author

Bektur Umarbaev hrafnulf13@gmail.com

Date

4.3.2 Macro Definition Documentation

4.3.2.1 __list_for_each

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

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

4.3.2.2 container of

(type *)((char *) __mptr - offsetof(type, member));})

Casts a member of a structure out to the containing structure

Parameters

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

4.3.2.3 hlist_for_each

Value:

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

4.3.2.4 hlist_for_each_entry

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

4.3.2.5 hlist_for_each_entry_continue

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

4.3.2.6 hlist for each entry from

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

4.3 list.h File Reference 23

4.3.2.7 hlist_for_each_entry_safe

hlist_for_each_entry_safe - iterate over list of given type safe against removal of list entry @tpos: the type * to use as a loop counter. @pos: the &struct hlist_node to use as a loop counter.

: another &struct hlist_node to use as temporary storage @head: the head for your list. @member: the name of the hlist_node within the struct.

4.3.2.8 hlist_for_each_safe

4.3.2.9 INIT_LIST_HEAD

4.3.2.10 list_entry

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

4.3.2.11 list_for_each

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

4.3.2.12 list_for_each_entry

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

4.3.2.13 list_for_each_entry_continue

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

4.3.2.14 list_for_each_entry_reverse

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

4.3 list.h File Reference 25

4.3.2.15 list_for_each_entry_safe

list_for_each_entry_safe - iterate over list of given type safe against removal of list entry @pos: the type * to use as a loop counter.

: another type * to use as temporary storage @head: the head for your list. @member: the name of the list_struct within the struct.

4.3.2.16 list for each entry safe continue

list_for_each_entry_safe_continue - iterate over list of given type continuing after existing point safe against removal of list entry @pos: the type * to use as a loop counter.

: another type * to use as temporary storage @head: the head for your list. @member: the name of the list_struct within the struct.

4.3.2.17 list_for_each_entry_safe_reverse

list_for_each_entry_safe_reverse - iterate backwards over list of given type safe against removal of list entry @pos: the type * to use as a loop counter.

: another type * to use as temporary storage @head: the head for your list. @member: the name of the list_struct within the struct.

4.3.2.18 list_for_each_prev

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

4.3.2.19 list_for_each_safe

list_for_each_safe - iterate over a list safe against removal of list entry @pos: the &struct list_head to use as a loop counter.

: another &struct list_head to use as temporary storage @head: the head for your list.

4.3.2.20 list_prepare_entry

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

4.3.2.21 offsetof

Get offset of a member

4.4 list.h 27

4.4 list.h

Go to the documentation of this file.

```
32 #ifndef _LINUX_LIST_H
33 #define _LINUX_LIST_H
35 #include <stdio.h>
40
44 #define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
4.5
55
           (type *)( (char *) __mptr - offsetof(type, member) );})
59 /*
60
   \star These are non-NULL pointers that will result in page faults
61 \star under normal circumstances, used to verify that nobody uses
62 * non-initialized list entries.
63
64 #define LIST_POISON1 ((void *) 0x00100100)
65 #define LIST_POISON2 ((void *) 0x00200200)
66
76 struct list_head {
      struct list_head *next, *prev;
78 };
80 #define LIST_HEAD_INIT(name) { &(name), &(name) }
81
82 #define LIST_HEAD(name) \
      struct list_head name = LIST_HEAD_INIT(name)
8.3
84
85 #define INIT_LIST_HEAD(ptr) do { \
       (ptr)->next = (ptr); (ptr)->prev = (ptr); \
87 } while (0)
88
89 /*
90 * Insert a new entry between two known consecutive entries.
91 *
92 * This is only for internal list manipulation where we know
93 \star the prev/next entries already!
94 */
95 static inline void __list_add(struct list_head *new,
                     struct list_head *prev,
struct list_head *next)
96
98 {
99
      next->prev = new;
100
        new->next = next;
101
       new->prev = prev;
        prev->next = new;
103 }
104
113 static inline void list_add(struct list_head *new, struct list_head *head)
114 {
115
        __list_add(new, head, head->next);
116 }
117
126 static inline void list_add_tail(struct list_head *new, struct list_head *head)
127 {
128
        __list_add(new, head->prev, head);
129 }
130
131
132 /*
133 * Delete a list entry by making the prev/next entries
134 * point to each other.
135 *
136 \star This is only for internal list manipulation where we know
137 * the prev/next entries already!
138 */
139 static inline void __list_del(struct list_head * prev, struct list_head * next)
140 {
        next->prev = prev;
prev->next = next;
141
142
143 }
144
151 static inline void list_del(struct list_head *entry)
152 {
153
          _list_del(entry->prev, entry->next);
        entry->next = LIST_POISON1;
entry->prev = LIST_POISON2;
154
155
156 }
157
158
164 static inline void list_del_init(struct list_head *entry)
```

```
165 {
          _list_del(entry->prev, entry->next);
166
167
        INIT_LIST_HEAD(entry);
168 }
169
175 static inline void list_move(struct list_head *list, struct list_head *head)
176 {
177
              _list_del(list->prev, list->next);
178
            list_add(list, head);
179 }
180
186 static inline void list_move_tail(struct list_head *list,
187
                      struct list head *head)
188 {
189
              _list_del(list->prev, list->next);
190
            list_add_tail(list, head);
191 }
192
197 static inline int list_empty(const struct list_head *head)
198 {
199
        return head->next == head;
200 }
2.01
202 static inline void __list_splice(struct list_head *list,
203
                     struct list_head *head)
204 {
205
        struct list_head *first = list->next;
206
        struct list_head *last = list->prev;
207
        struct list_head *at = head->next;
208
209
       first->prev = head:
210
       head->next = first;
211
212
        last->next = at;
213
        at->prev = last;
214 }
215
221 static inline void list_splice(struct list_head *list, struct list_head *head)
222 {
223
        if (!list_empty(list))
224
            __list_splice(list, head);
225 }
226
234 static inline void list_splice_init(struct list_head *list,
                        struct list_head *head)
236 {
237
        if (!list_empty(list)) {
238
              _list_splice(list, head);
            INIT_LIST_HEAD(list);
239
240
241 }
242
249 #define list_entry(ptr, type, member) \
250
      container_of(ptr, type, member)
251
258 #define list_for_each(pos, head) \
    for (pos = (head) ->next; pos != (head); \
          pos = pos->next)
260
261
      efine __list_for_each(pos, head) \
  for (pos = (head) ->next; pos != (head); pos = pos->next)
272 #define _
273
274
280 #define list_for_each_prev(pos, head) \
    for (pos = (head) ->prev; prefetch (pos->prev), pos != (head); \
282
                pos = pos->prev)
283
290 #define list_for_each_safe(pos, n, head) \
291    for (pos = (head) ->next, n = pos->next; pos != (head); \

            pos = n, n = pos -> next)
292
300 #define list_for_each_entry(pos, head, member)
301
      for (pos = list_entry((head)->next, typeof(*pos), member);
             &pos->member != (head);
302
             pos = list_entry(pos->member.next, typeof(*pos), member))
303
304
311 #define list_for_each_entry_reverse(pos, head, member)
      for (pos = list_entry((head)->prev, typeof(*pos), member);
312
313
             &pos->member != (head);
314
             pos = list_entry(pos->member.prev, typeof(*pos), member))
315
323 #define list_prepare_entry(pos, head, member) \
       ((pos) ? : list_entry(head, typeof(*pos), member))
324
325
333 #define list_for_each_entry_continue(pos, head, member)
334
      for (pos = list_entry(pos->member.next, typeof(*pos), member); \
             &pos->member != (head):
335
             pos = list_entry(pos->member.next, typeof(*pos), member))
336
```

4.4 list.h 29

```
337
345 #define list_for_each_entry_safe(pos, n, head, member)
346
        for (pos = list_entry((head) ->next, typeof(*pos), member),
           n = list_entry(pos->member.next, typeof(*pos), member); \
347
             &pos->member != (head);
pos = n, n = list_entry(n->member.next, typeof(*n), member))
348
349
350
359 #define list_for_each_entry_safe_continue(pos, n, head, member)
360
       for (pos = list_entry(pos->member.next, typeof(*pos), member),
361
            n = list\_entry(pos->member.next, typeof(*pos), member);
             &pos->member != (head);
362
363
             pos = n, n = list_entry(n->member.next, typeof(*n), member))
364
373 #define list_for_each_entry_safe_reverse(pos, n, head, member)
374
        for (pos = list_entry((head)->prev, typeof(*pos), member),
375
            n = list_entry(pos->member.prev, typeof(*pos), member); \
             &pos->member != (head);
pos = n, n = list_entry(n->member.prev, typeof(*n), member))
376
377
378
379
380
381
382 /*
383 \star Double linked lists with a single pointer list head.
384 * Mostly useful for hash tables where the two pointer list head is
   * too wasteful.
386 \star You lose the ability to access the tail in O(1).
387 */
388
389 struct hlist head {
390
       struct hlist node *first;
391 };
392
393 struct hlist_node {
394
       struct hlist_node *next, **pprev;
395 };
396
397 #define HLIST_HEAD_INIT { .first = NULL }
398 #define HLIST_HEAD(name) struct hlist_head name = { .first = NULL }
399 #define INIT_HLIST_HEAD(ptr) ((ptr)->first = NULL)
400 #define INIT_HLIST_NODE(ptr) ((ptr)->next = NULL, (ptr)->pprev = NULL)
401
402 static inline int hlist unhashed(const struct hlist node *h)
403 {
404
        return !h->pprev;
405 }
406
407 static inline int hlist_empty(const struct hlist_head \starh)
408 {
409
        return !h->first;
410 }
411
412 static inline void __hlist_del(struct hlist_node *n)
413 {
        struct hlist_node *next = n->next;
414
415
        struct hlist node **pprev = n->pprev;
416
        *pprev = next;
417
        if (next)
418
           next->pprev = pprev;
419 }
420
421 static inline void hlist del(struct hlist node *n)
422 {
423
          _hlist_del(n);
424
        n->next = LIST_POISON1;
425
        n->pprev = LIST_POISON2;
426 }
427
428
429 static inline void hlist_del_init(struct hlist_node *n)
430 {
431
        if (n->pprev)
432
              _hlist_del(n);
            INIT_HLIST_NODE(n);
433
434
        }
435 }
436
437 static inline void hlist_add_head(struct hlist_node *n, struct hlist_head *h)
438 {
439
        struct hlist node *first = h->first;
        n->next = first;
440
441
        if (first)
442
            first->pprev = &n->next;
443
        h \rightarrow first = n;
444
        n->pprev = &h->first;
445 }
446
```

```
447
449 /* next must be != NULL */
450 static inline void hlist_add_before(struct hlist_node *n,
4.5.1
                       struct hlist_node *next)
452 {
453
       n->pprev = next->pprev;
454
       n->next = next;
455
       next->pprev = &n->next;
456
       *(n->pprev) = n;
457 }
458
459 static inline void hlist_add_after(struct hlist_node *n,
460
                       struct hlist_node *next)
461 {
462
       next->next = n->next;
463
       n->next = next;
464
       next->pprev = &n->next;
465
466
       if (next->next)
467
           next->next->pprev = &next->next;
468 }
469
470
472 #define hlist_entry(ptr, type, member) container_of(ptr,type,member)
473
474 #define hlist_for_each(pos, head) \
      for (pos = (head)->first; pos && ({ prefetch(pos->next); 1; }); \
pos = pos->next)
475
476
477
481
489 #define hlist_for_each_entry(tpos, pos, head, member)
      for (pos = (head) ->first;
490
            pos && ({ prefetch(pos->next); 1;}) &&
492
            ({ tpos = hlist_entry(pos, typeof(*tpos), member); 1;}); \
493
            pos = pos->next)
494
501 #define hlist_for_each_entry_continue(tpos, pos, member)
      for (pos = (pos)->next;
502
           pos && ({ prefetch(pos->next); 1;}) && (
{ tpos = hlist_entry(pos, typeof(*tpos), member); 1;}); \
503
505
            pos = pos->next)
506
513 #define hlist_for_each_entry_from(tpos, pos, member)
       514
515
516
            pos = pos->next)
517
526 #define hlist_for_each_entry_safe(tpos, pos, n, head, member)
     for (pos = (head) ->first;
    pos && ({ n = pos->next; 1; }) &&
        ({ tpos = hlist_entry(pos, typeof(*tpos), member); 1;}); \
527
528
529
531
532
533 #endif
```

4.5 ums_lib.c File Reference

Contains implementations of the essential UMS library functions.

```
#include "ums_lib.h"
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <pthread.h>
#include <fcntl.h>
#include <errno.h>
#include <sched.h>
#include <unistd.h>
```

Macros

- #define _GNU_SOURCE
- #define create_list_params(size) (list_params_t*)malloc(sizeof(list_params_t) + size * sizeof(ums_wid_t))

Functions

• int open device ()

Opens UMS device Uses mutex to protect a shared resource from simultaneous access by multiple threads.

• int close device ()

Closes UMS device Uses mutex to protect a shared resource from simultaneous access by multiple threads.

• int ums enter ()

Requests UMS kernel module to manage current process

• int ums exit ()

Requests UMS kernel module to finish management of the current process

ums_clid_t ums_create_completion_list ()

Requests UMS kernel module to create a completion lists

ums_wid_t ums_create_worker_thread (ums_clid_t clid, unsigned long stack_size, void(*entry_point)(void *),
 void *args)

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

• ums sid tums create scheduler (ums clid t clid, void(*entry point)())

Wrapper function that creates pthreds which eventually request UMS kernel module to create a scheduler UMS library uses pthread library to create process threads that will become scheduler threads. Each pthread jumps to ums_enter_scheduling_mode() function and requests UMS kernel module to create a scheduler by passing scheduler_params. After successful creation of the scheduler by the UMS kernel module, created pthread becomes scheduler. It starts scheduler work by jumping to the entry point assigned by a user and stays there until ums_exit_scheduling_mode() is called. Here list_params is also created for the future calls of ums_dequeue_completion_list_items() by a scheduler (since in this stage the completion list has been fully populated and cannot be modified later).

void * ums enter scheduling mode (void *args)

Actual function that is called by a pthread to request the UMS kernel module in order create a scheduler and assign a completion list to it Additionally assigns a CPU core on which the scheduler will operate based on available cores.

• int ums exit scheduling mode ()

Called by a scheduler to signal the UMS kernel module about the completion of scheduling mode Restores instruction, stack and base pointers to return back to ums_enter_scheduling_mode() function to perform pthread_exit()

int ums_execute_thread (ums_wid_t wid)

Called by a scheduler to request UMS kernel module to execute a worker thread with specific ID

• int ums_thread_yield (worker_status_t status)

Called by a worker thread to pause or complete the execution Depending on the value of the argument, the function will:

• int ums thread pause ()

Called by a worker thread to pause the execution Wrapper that calls ums_thread_yield() with an argument PAUSE.

int ums_thread_exit ()

Called by a worker thread to complete the execution Wrapper that calls ums_thread_yield() with an argument FINISH.

list_params_t * ums_dequeue_completion_list_items ()

Called by a scheduler to request UMS kernel module to provide a list of available worker threads that can be scheduled The function passes a global list_params from the ums_completion_list_node structure to the UMS kernel module The kernel module populates the structure with the list of available workers and sets the number of those available workers. Each scheduler has own copy of the list_params, but can notify other scheduler about the state of the completion list (if shared) by updating its' state. Thus other schedulers do not have to perform ioctl call, just update their own list_params and set its' state to FINISHED.

ums_wid_t ums_get_next_worker_thread (list_params_t *list)

Called by a scheduler, after performing ums_dequeue_completion_list_items(), to find a next available worker thread from the completion list This function always has to be run after calling ums_dequeue_completion_list_items(), since it will populate the list in the correct way to be processed Passing a manually created list parameter will result in undefined behaviour.

• int cleanup ()

Performs a cleanup by deleting all the data structures allocated by the library

ums completion list node t*check if completion list exists (ums clid t clid)

Checks if the completion list with a passed ID exists or not

• ums worker t * check if worker exists (ums wid t wid)

Checks if the worker thread with a passed ID exists or not

ums scheduler t * check if scheduler exists ()

Checks if the scheduler for the current pthread exists or not

- __attribute__ ((constructor))
- __attribute__ ((destructor))

Variables

- int ums dev = -UMS ERROR
- pthread mutex t ums_mutex = PTHREAD MUTEX INITIALIZER
- · ums_completion_list_t completion_lists
- ums_worker_list_t workers
- · ums scheduler list t schedulers
- __thread ums_clid_t completion_list_id

4.5.1 Detailed Description

Contains implementations of the essential UMS library functions.

Copyright (C) 2021 Bektur Umarbaev hrafnulf13@gmail.com

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

UMS 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 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 library. If not, see http←://www.gnu.org/licenses/.

Author

Bektur Umarbaev hrafnulf13@gmail.com

Date

4.5.2 Function Documentation

4.5.2.1 check_if_completion_list_exists()

Checks if the completion list with a passed ID exists or not

Parameters

```
clid Completion list ID
```

Returns

returns a pointer to the existing completion list structure if it exists, NULL otherwise

4.5.2.2 check_if_scheduler_exists()

```
ums_scheduler_t * check_if_scheduler_exists ( )
```

Checks if the scheduler for the current pthread exists or not

Returns

returns a pointer to the existing scheduler structure if it exists, NULL otherwise

4.5.2.3 check_if_worker_exists()

Checks if the worker thread with a passed ID exists or not

Parameters

```
wid Worker thread ID
```

Returns

returns a UMS_SUCCESS if worker thread exists, UMS_ERROR_WORKER_NOT_FOUND otherwise

4.5.2.4 cleanup()

```
int cleanup ( )
```

Performs a cleanup by deleting all the data structures allocated by the library

Returns

returns UMS_SUCCESS when succesful or UMS_ERROR if there are any errors

4.5.2.5 close_device()

```
int close_device ( )
```

Closes UMS device Uses mutex to protect a shared resource from simultaneous access by multiple threads.

Returns

returns UMS_SUCCESS when succesful or UMS_ERROR if there are any errors

4.5.2.6 open device()

```
int open_device ( )
```

Opens UMS device Uses mutex to protect a shared resource from simultaneous access by multiple threads.

Returns

returns UMS_SUCCESS when succesful or UMS_ERROR if there are any errors

4.5.2.7 ums create completion list()

```
ums_clid_t ums_create_completion_list ( )
```

Requests UMS kernel module to create a completion lists

Returns

returns Completion list ID

4.5.2.8 ums_create_scheduler()

```
ums_sid_t ums_create_scheduler (
          ums_clid_t clid,
          void(*)() entry_point )
```

Wrapper function that creates pthreds which eventually request UMS kernel module to create a scheduler UMS library uses pthread library to create process threads that will become scheduler threads. Each pthread jumps to ums_enter_scheduling_mode() function and requests UMS kernel module to create a scheduler by passing scheduler_params. After successful creation of the scheduler by the UMS kernel module, created pthread becomes scheduler. It starts scheduler work by jumping to the entry point assigned by a user and stays there until ums_exit_scheduling_mode() is called. Here list_params is also created for the future calls of ums_dequeue_completion_list_items() by a scheduler (since in this stage the completion list has been fully populated and cannot be modified later).

Parameters

clid	ID of the completion list that is assigned to the scheduler	
entry_point	Function pointer and an entry point set by a user, that serves as a starting point of the scheduler.	
	It is a scheduling function that determines the next thread to be scheduled	

Returns

returns Scheduler ID

4.5.2.9 ums_create_worker_thread()

```
ums_wid_t ums_create_worker_thread (
    ums_clid_t clid,
    unsigned long stack_size,
    void(*)(void *) entry_point,
    void * args )
```

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

Parameters

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

Returns

returns Worker ID

4.5.2.10 ums_dequeue_completion_list_items()

```
list_params_t * ums_dequeue_completion_list_items ( )
```

Called by a scheduler to request UMS kernel module to provide a list of available worker threads that can be scheduled The function passes a global list_params from the ums_completion_list_node structure to the UMS kernel module The kernel module populates the structure with the list of available workers and sets the number of those available workers. Each scheduler has own copy of the list_params, but can notify other scheduler about the state of the completion list (if shared) by updating its' state. Thus other schedulers do not have to perform ioctl call, just update their own list_params and set its' state to FINISHED.

Returns

returns the pointer to a shared list_params structure which contains an array of available workers that can be scheduled

4.5.2.11 ums_enter()

```
int ums_enter ( )
```

Requests UMS kernel module to manage current process

Returns

returns UMS_SUCCESS when succesful or UMS_ERROR if there are any errors

4.5.2.12 ums_enter_scheduling_mode()

Actual function that is called by a pthread to request the UMS kernel module in order create a scheduler and assign a completion list to it Additionally assigns a CPU core on which the scheduler will operate based on available cores.

Parameters

args

Pointer to scheduler params that is passed in order to create a scheduler

Returns

4.5.2.13 ums_execute_thread()

```
int ums_execute_thread (
    ums_wid_t wid )
```

Called by a scheduler to request UMS kernel module to execute a worker thread with specific ID

Parameters

wid ID of the worker thread that to be executed

Returns

4.5.2.14 ums_exit()

```
int ums_exit ( )
```

Requests UMS kernel module to finish management of the current process

Returns

returns UMS_SUCCESS when succesful or UMS_ERROR if there are any errors

4.5.2.15 ums_exit_scheduling_mode()

```
int ums_exit_scheduling_mode ( )
```

Called by a scheduler to signal the UMS kernel module about the completion of scheduling mode Restores instruction, stack and base pointers to return back to ums_enter_scheduling_mode() function to perform pthread_exit()

Returns

4.5.2.16 ums_get_next_worker_thread()

Called by a scheduler, after performing ums_dequeue_completion_list_items(), to find a next available worker thread from the completion list This function always has to be run after calling ums_dequeue_completion_list_items(), since it will populate the list in the correct way to be processed Passing a manually created list parameter will result in undefined behaviour.

Parameters

list

List parameter that is created after ums_dequeue_completion_list_items() call and contains the list of available workers

Returns

returns a next available worker thread that can be scheduled, or error values otherwise

4.5.2.17 ums_thread_exit()

```
int ums_thread_exit ( )
```

Called by a worker thread to complete the execution Wrapper that calls $ums_thread_yield()$ with an argument FINISH.

Returns

4.5.2.18 ums_thread_pause()

```
int ums_thread_pause ( )
```

Called by a worker thread to pause the execution Wrapper that calls ums_thread_yield() with an argument PAUSE.

Returns

4.5.2.19 ums_thread_yield()

Called by a worker thread to pause or complete the execution Depending on the value of the argument, the function will:

- Remove the worker thread from the list of worker threads that can be scheduled, thus completes the execution;
- · Push it back to the list of available worker thread, thus pauses its' execution and can be rescheduled later.

Parameters

status defines the status of the execution flow of the worker thread (passing PAUSE will pause the execution, when FINISH will complete it)

Returns

4.5.3 Variable Documentation

4.5.3.1 completion_lists

```
{\tt ums\_completion\_list\_t\ completion\_lists}
```

Initial value:

```
= {
    .list = LIST_HEAD_INIT(completion_lists.list),
    .count = 0
}
```

4.5.3.2 schedulers

```
ums_scheduler_list_t schedulers

Initial value:
= {
    .list = LIST_HEAD_INIT(schedulers.list),
    .count = 0
}
```

4.5.3.3 workers

```
ums_worker_list_t workers

Initial value:
= {
    .list = LIST_HEAD_INIT(workers.list),
    .count = 0
```

4.6 ums_lib.h File Reference

The header that contains essential UMS library functions and has to be included by the user in order to use the UMS library.

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

Data Structures

· struct ums_completion_list

The list of the completion lists created by the process

• struct ums_completion_list_node

Represents a node in the ums_completion_list

struct ums_worker_list

The list of the worker threads created by the process

· struct ums worker

Represents a node in the ums_worker_list

struct ums_scheduler_list

The list of the schedulers created by the process

• struct ums_scheduler

Represents a node in the ums_scheduler_list

Macros

- #define UMS_DEVICE "/dev/ums"
- #define init(type) (type*)malloc(sizeof(type))
- #define delete(val) free(val)

Typedefs

· typedef struct ums completion list ums completion list t

The list of the completion lists created by the process

• typedef struct ums_completion_list_node ums_completion_list_node_t

Represents a node in the ums_completion_list

typedef struct ums_worker ums_worker_t

Represents a node in the ums_worker_list

typedef struct ums_worker_list ums_worker_list_t

The list of the worker threads created by the process

typedef struct ums_scheduler_list ums_scheduler_list_t

The list of the schedulers created by the process

• typedef struct ums_scheduler ums_scheduler_t

Represents a node in the ums_scheduler_list

Functions

• int ums_enter ()

Requests UMS kernel module to manage current process

• int ums exit ()

Requests UMS kernel module to finish management of the current process

ums_clid_t ums_create_completion_list ()

Requests UMS kernel module to create a completion lists

ums_wid_t ums_create_worker_thread (ums_clid_t clid, unsigned long stack_size, void(*entry_point)(void *),
 void *args)

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

- ums_sid_t ums_create_scheduler (ums_clid_t clid, void(*entry_point)(void *))
- void * ums_enter_scheduling_mode (void *args)

Actual function that is called by a pthread to request the UMS kernel module in order create a scheduler and assign a completion list to it Additionally assigns a CPU core on which the scheduler will operate based on available cores.

• int ums exit scheduling mode ()

Called by a scheduler to signal the UMS kernel module about the completion of scheduling mode Restores instruction, stack and base pointers to return back to ums_enter_scheduling_mode() function to perform pthread_exit()

• int ums execute thread (ums wid t wid)

Called by a scheduler to request UMS kernel module to execute a worker thread with specific ID

- int ums_thread_yield ()
- int ums_thread_pause ()

Called by a worker thread to pause the execution Wrapper that calls ums thread yield() with an argument PAUSE.

int ums_thread_exit ()

Called by a worker thread to complete the execution Wrapper that calls ums_thread_yield() with an argument FINISH.

• list_params_t * ums_dequeue_completion_list_items ()

Called by a scheduler to request UMS kernel module to provide a list of available worker threads that can be scheduled The function passes a global list_params from the ums_completion_list_node structure to the UMS kernel module The kernel module populates the structure with the list of available workers and sets the number of those available workers. Each scheduler has own copy of the list_params, but can notify other scheduler about the state of the completion list (if shared) by updating its' state. Thus other schedulers do not have to perform ioctl call, just update their own list_params and set its' state to FINISHED.

ums_wid_t ums_get_next_worker_thread (list_params_t *list)

Called by a scheduler, after performing ums_dequeue_completion_list_items(), to find a next available worker thread from the completion list This function always has to be run after calling ums_dequeue_completion_list_items(), since it will populate the list in the correct way to be processed Passing a manually created list parameter will result in undefined behaviour.

• int open_device ()

Opens UMS device Uses mutex to protect a shared resource from simultaneous access by multiple threads.

• int close device ()

Closes UMS device Uses mutex to protect a shared resource from simultaneous access by multiple threads.

• int cleanup ()

Performs a cleanup by deleting all the data structures allocated by the library

• ums_completion_list_node_t * check_if_completion_list_exists (ums_clid_t clid)

Checks if the completion list with a passed ID exists or not

ums_worker_t * check_if_worker_exists (ums_wid_t wid)

Checks if the worker thread with a passed ID exists or not

• ums scheduler t * check if scheduler exists ()

Checks if the scheduler for the current pthread exists or not

4.6.1 Detailed Description

The header that contains essential UMS library functions and has to be included by the user in order to use the UMS library.

Copyright (C) 2021 Bektur Umarbaev hrafnulf13@gmail.com

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

UMS 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 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 library. If not, see http://www.gnu.org/licenses/.

Author

Bektur Umarbaev hrafnulf13@gmail.com

Date

4.6.2 Function Documentation

4.6.2.1 check_if_completion_list_exists()

Checks if the completion list with a passed ID exists or not

Parameters

```
clid Completion list ID
```

Returns

returns a pointer to the existing completion list structure if it exists, NULL otherwise

4.6.2.2 check_if_scheduler_exists()

```
ums_scheduler_t * check_if_scheduler_exists ( )
```

Checks if the scheduler for the current pthread exists or not

Returns

returns a pointer to the existing scheduler structure if it exists, NULL otherwise

4.6.2.3 check_if_worker_exists()

```
ums_worker_t * check_if_worker_exists (
    ums_wid_t wid )
```

Checks if the worker thread with a passed ID exists or not

Parameters

```
wid Worker thread ID
```

Returns

returns a UMS_SUCCESS if worker thread exists, UMS_ERROR_WORKER_NOT_FOUND otherwise

4.6.2.4 cleanup()

```
int cleanup ( )
```

Performs a cleanup by deleting all the data structures allocated by the library

Returns

returns UMS_SUCCESS when succesful or UMS_ERROR if there are any errors

4.6.2.5 close_device()

```
int close_device ( )
```

Closes UMS device Uses mutex to protect a shared resource from simultaneous access by multiple threads.

Returns

returns UMS_SUCCESS when succesful or UMS_ERROR if there are any errors

4.6.2.6 open_device()

```
int open_device ( )
```

Opens UMS device Uses mutex to protect a shared resource from simultaneous access by multiple threads.

Returns

returns UMS_SUCCESS when succesful or UMS_ERROR if there are any errors

4.6.2.7 ums_create_completion_list()

```
ums_clid_t ums_create_completion_list ( )
```

Requests UMS kernel module to create a completion lists

Returns

returns Completion list ID

4.6.2.8 ums_create_worker_thread()

```
ums_wid_t ums_create_worker_thread (
          ums_clid_t clid,
          unsigned long stack_size,
          void(*)(void *) entry_point,
          void * args )
```

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

Parameters

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

Returns

returns Worker ID

4.6.2.9 ums_dequeue_completion_list_items()

```
list_params_t * ums_dequeue_completion_list_items ( )
```

Called by a scheduler to request UMS kernel module to provide a list of available worker threads that can be scheduled The function passes a global list_params from the ums_completion_list_node structure to the UMS kernel module The kernel module populates the structure with the list of available workers and sets the number of those available workers. Each scheduler has own copy of the list_params, but can notify other scheduler about the state of the completion list (if shared) by updating its' state. Thus other schedulers do not have to perform ioctl call, just update their own list_params and set its' state to FINISHED.

Returns

returns the pointer to a shared list_params structure which contains an array of available workers that can be scheduled

4.6.2.10 ums_enter()

```
int ums_enter ( )
```

Requests UMS kernel module to manage current process

Returns

returns UMS_SUCCESS when succesful or UMS_ERROR if there are any errors

4.6.2.11 ums_enter_scheduling_mode()

Actual function that is called by a pthread to request the UMS kernel module in order create a scheduler and assign a completion list to it Additionally assigns a CPU core on which the scheduler will operate based on available cores.

Parameters

args

Pointer to scheduler params that is passed in order to create a scheduler

Returns

4.6.2.12 ums_execute_thread()

```
int ums_execute_thread (
     ums_wid_t wid )
```

Called by a scheduler to request UMS kernel module to execute a worker thread with specific ID

Parameters

wid

ID of the worker thread that to be executed

Returns

4.6.2.13 ums_exit()

```
int ums_exit ( )
```

Requests UMS kernel module to finish management of the current process

Returns

returns UMS_SUCCESS when succesful or UMS_ERROR if there are any errors

4.6.2.14 ums_exit_scheduling_mode()

```
int ums_exit_scheduling_mode ( )
```

Called by a scheduler to signal the UMS kernel module about the completion of scheduling mode Restores instruction, stack and base pointers to return back to ums_enter_scheduling_mode() function to perform pthread_exit()

Returns

4.6.2.15 ums_get_next_worker_thread()

Called by a scheduler, after performing ums_dequeue_completion_list_items(), to find a next available worker thread from the completion list This function always has to be run after calling ums_dequeue_completion_list_items(), since it will populate the list in the correct way to be processed Passing a manually created list parameter will result in undefined behaviour.

Parameters

list

List parameter that is created after ums_dequeue_completion_list_items() call and contains the list of available workers

Returns

returns a next available worker thread that can be scheduled, or error values otherwise

4.6.2.16 ums_thread_exit()

```
int ums_thread_exit ( )
```

Called by a worker thread to complete the execution Wrapper that calls ums_thread_yield() with an argument FINISH.

Returns

4.6.2.17 ums_thread_pause()

```
int ums_thread_pause ( )
```

 $Called \ by \ a \ worker \ thread \ to \ pause \ the \ execution \ Wrapper \ that \ calls \ ums_thread_yield() \ with \ an \ argument \ {\tt PAUSE}.$

Returns

4.7 ums lib.h

Go to the documentation of this file.

```
29 #pragma once
30
31 #include "const.h"
32 #include "list.h"
33 #include <pthread.h>
34
35 #define UMS_DEVICE "/dev/ums"
36
37 typedef struct ums_completion_list ums_completion_list_t;
38 typedef struct ums_completion_list_node ums_completion_list_node_t;
39 typedef struct ums_worker ums_worker_t;
40 typedef struct ums_worker_list ums_worker_list_t;
41 typedef struct ums_scheduler_list ums_scheduler_list_t;
42 typedef struct ums_scheduler ums_scheduler_t;
44 int ums_enter();
45 int ums_exit();
47 ums_clid_t ums_create_completion_list();
48 ums_wid_t ums_create_worker_thread(ums_clid_t clid, unsigned long stack_size, void (*entry_point)(void
      *), void *args);
49 ums_sid_t ums_create_scheduler(ums_clid_t clid, void (*entry_point)(void *));
50 void *ums_enter_scheduling_mode(void *args);
51 int ums_exit_scheduling_mode();
52 int ums_execute_thread(ums_wid_t wid);
53 int ums_thread_yield();
54 int ums_thread_pause();
55 int ums_thread_exit();
56 list_params_t *ums_dequeue_completion_list_items();
57 ums_wid_t ums_get_next_worker_thread(list_params_t *list);
59 int open device():
60 int close device();
61 int cleanup();
62 ums_completion_list_node_t *check_if_completion_list_exists(ums_clid_t clid);
63 ums_worker_t *check_if_worker_exists(ums_wid_t wid);
64 ums_scheduler_t *check_if_scheduler_exists();
65
70 typedef struct ums_completion_list {
      struct list_head list;
71
      unsigned int count;
73 } ums_completion_list_t;
74
79 typedef struct ums_completion_list_node {
80
     ums_clid_t clid;
       state_t state;
81
      unsigned int worker_count;
     struct list_head list;
list_params_t *list_params;
84
85 } ums_completion_list_node_t;
86
91 typedef struct ums_worker_list {
    struct list_head list;
unsigned int count;
94 } ums_worker_list_t;
95
100 typedef struct ums_worker {
       ums_wid_t wid;
101
        state_t state;
        struct list_head list;
103
104
       worker_params_t *worker_params;
105 } ums_worker_t;
106
111 typedef struct ums_scheduler_list {
112
       struct list_head list;
        unsigned int count;
113
114 } ums_scheduler_list_t;
115
120 typedef struct ums_scheduler {
       struct list_head list;
121
122
        pthread t tid:
123
        ums_wid_t wid;
124
        scheduler_params_t *sched_params;
125
        list_params_t *list_params;
126 } ums_scheduler_t;
127
128 #define init(type) (type*)malloc(sizeof(type))
129 #define delete(val) free(val)
130 #define create_list_params(size) (list_params_t*)malloc(sizeof(list_params_t) + size *
       sizeof(ums_wid_t))
```

Index

```
__list_for_each
                                                               worker_params, 14
     list.h, 21
                                                          hlist for each
check_if_completion_list_exists
                                                               list.h, 21
     ums lib.c, 33
                                                          hlist_for_each_entry
     ums lib.h, 43
                                                               list.h, 22
check_if_scheduler_exists
                                                          hlist_for_each_entry_continue
     ums_lib.c, 34
                                                               list.h, 22
     ums_lib.h, 43
                                                          hlist_for_each_entry_from
check if worker exists
                                                               list.h, 22
     ums_lib.c, 34
                                                          hlist_for_each_entry_safe
     ums lib.h, 43
                                                               list.h, 22
cleanup
                                                          hlist for each safe
     ums lib.c, 34
                                                               list.h, 23
     ums lib.h, 44
                                                          hlist head, 5
clid
                                                          hlist_node, 5
     scheduler params, 7
                                                          IDLE
     ums completion list node, 9
                                                               const.h, 18
     worker_params, 14
                                                          INIT LIST HEAD
close_device
                                                               list.h, 23
     ums_lib.c, 34
     ums lib.h, 44
                                                          list.h, 19
completion lists
                                                                 _list_for_each, 21
     ums lib.c, 39
                                                               container_of, 21
const.h, 15
                                                               hlist for each, 21
     FINISH. 18
                                                               hlist for each entry, 22
     FINISHED, 18
                                                               hlist_for_each_entry_continue, 22
     IDLE, 18
                                                               hlist_for_each_entry_from, 22
     PAUSE, 18
                                                               hlist for each entry safe, 22
     RUNNING, 18
                                                               hlist for each safe, 23
     state, 17
                                                               INIT_LIST_HEAD, 23
     worker_status, 18
                                                               list_entry, 23
container of
                                                               list_for_each, 23
     list.h, 21
                                                               list_for_each_entry, 24
core id
                                                               list_for_each_entry_continue, 24
     scheduler params, 7
                                                               list_for_each_entry_reverse, 24
count
                                                               list for each entry safe, 24
     ums_completion_list, 8
                                                               list for each entry safe continue, 25
     ums_scheduler_list, 11
                                                               list_for_each_entry_safe_reverse, 25
     ums_worker_list, 13
                                                               list_for_each_prev, 25
                                                               list for each safe, 26
entry_point
                                                               list_prepare_entry, 26
     scheduler params, 8
                                                               offsetof, 26
     worker_params, 14
                                                          list_entry
FINISH
                                                               list.h, 23
     const.h, 18
                                                          list_for_each
FINISHED
                                                               list.h, 23
     const.h, 18
                                                          list for each entry
function args
                                                               list.h, 24
```

50 INDEX

list_for_each_entry_continue	tid
list.h, 24	ums_scheduler, 11
list_for_each_entry_reverse	
list.h, 24	ums_completion_list, 8
list_for_each_entry_safe	count, 8
list.h, 24	ums_completion_list_node, 9
list_for_each_entry_safe_continue	clid, 9
list.h, 25	list_params, 9
list_for_each_entry_safe_reverse	state, 9
list.h, 25	worker_count, 10
list_for_each_prev	ums_create_completion_list
list.h, 25	ums_lib.c, 35
list_for_each_safe	ums_lib.h, 44
list.h, 26	ums_create_scheduler
list_head, 5	ums_lib.c, 35
list_params, 6	ums_create_worker_thread
size, 6	ums_lib.c, 36
state, 6	ums_lib.h, 44
ums_completion_list_node, 9	ums_dequeue_completion_list_items
ums_scheduler, 10	ums_lib.c, 36
worker_count, 6	ums_lib.h, 45
workers, 7	ums_enter
list_prepare_entry	ums_lib.c, 36
list.h, 26	ums_lib.h, 45
	ums_enter_scheduling_mode
offsetof	ums_lib.c, 37
list.h, 26	ums_lib.h, 45
open_device	ums_execute_thread
ums_lib.c, 35	ums_lib.c, 37
ums_lib.h, 44	ums_lib.h, 46
	ums_exit
PAUSE	ums_lib.c, 37
const.h, 18	ums_lib.h, 46
DUNINING	ums_exit_scheduling_mode
RUNNING	ums_lib.c, 38
const.h, 18	ums_lib.h, 46
sched params	ums_get_next_worker_thread
ums scheduler, 10	ums_lib.c, 38
scheduler_params, 7	ums_lib.h, 46
clid, 7	ums_lib.c, 30
core id, 7	check_if_completion_list_exists, 33
_ ·	check_if_scheduler_exists, 34
entry_point, 8	check_if_worker_exists, 34
sid, 8	cleanup, 34
schedulers	close_device, 34
ums_lib.c, 40	completion_lists, 39
sid	open_device, 35
scheduler_params, 8	schedulers, 40
Size	ums_create_completion_list, 35
list_params, 6	ums_create_scheduler, 35
stack_addr	ums_create_worker_thread, 36
worker_params, 14	ums_dequeue_completion_list_items, 36
stack_size	ums_enter, 36
worker_params, 14	ums_enter_scheduling_mode, 37
state	ums_execute_thread, 37
const.h, 17	ums_exit, 37
list_params, 6	ums_exit_scheduling_mode, 38
ums_completion_list_node, 9	ums_get_next_worker_thread, 38
ums_worker, 12	ums_thread_exit, 38

INDEX 51

```
ums_thread_pause, 39
                                                           list_params, 7
    ums_thread_yield, 39
                                                           ums_lib.c, 40
    workers, 40
ums_lib.h, 40
    check_if_completion_list_exists, 43
    check if scheduler exists, 43
    check_if_worker_exists, 43
    cleanup, 44
    close device, 44
    open device, 44
    ums_create_completion_list, 44
    ums_create_worker_thread, 44
    ums_dequeue_completion_list_items, 45
    ums_enter, 45
    ums_enter_scheduling_mode, 45
    ums_execute_thread, 46
    ums exit, 46
    ums_exit_scheduling_mode, 46
    ums_get_next_worker_thread, 46
    ums_thread_exit, 47
    ums_thread_pause, 47
ums_scheduler, 10
    list_params, 10
    sched_params, 10
    tid, 11
    wid, 11
ums_scheduler_list, 11
    count, 11
ums thread exit
    ums_lib.c, 38
    ums_lib.h, 47
ums_thread_pause
    ums lib.c, 39
    ums_lib.h, 47
ums_thread_yield
    ums_lib.c, 39
ums_worker, 12
    state, 12
    wid, 12
    worker_params, 12
ums worker list, 13
    count, 13
wid
    ums scheduler, 11
    ums_worker, 12
worker_count
    list_params, 6
    ums_completion_list_node, 10
worker_params, 13
    clid, 14
    entry point, 14
    function args, 14
    stack_addr, 14
    stack_size, 14
    ums worker, 12
worker_status
    const.h, 18
workers
```