Linux User-Mode Scheduling

Release v1.0.0

Arella Matteo

Oct 28, 2021

CONTENTS

CHAPTER ONE

OVERVIEW

User-mode scheduling (UMS) is a lightweight mechanism that applications can use to schedule their own threads. The ability to switch between threads in user mode makes UMS more efficient than thread pools for managing large numbers of short-duration work items that require few system calls.

To take advantage of UMS, an application must implement a scheduler component that manages the application's UMS threads and determines when they should run.

The implementation of a User-mode scheduling is inspired from the UMS specification available in the Windows kernel.

CHAPTER

TWO

BUILDING

2.1 Minimal Requirements

- GNU C >= 5.1
- GNU make >= 3.81
- libtool
- pkg-config
- linux-headers
- check[1]
- sphinx[2]
- doxygen[?]
- breathe[?]
- sphinx_rtd_theme[?]
- latex[3]

2.2 Development Requirements

The following dependencies are needed in case of development build:

- autotools-dev
- autoconf
- automake
- git
- cppcheck[4]

[1] Optional: needed only for userspace library tests
[2] Optional: needed only to build the documentation
[3] Optional: needed only to build the pdf documentation
[4] Optional: needed only for userspace library static code analysis

2.3 Build

```
$ ./configure [OPTION]... [VAR=VALUE]... && make [TARGET]...
```

To assign environment variables (e.g., CC, CFLAGS...), specify them as VAR=VALUE. See below for descriptions of some of the useful variables.

2.3.1 Options

--enable-examples	Enable examples build
--enable-benchmark	Enable benchmark build
--enable-docs	Enable documentation build
--prefix=PREFIX	Install architecture-independent files in PREFIX; default to /usr/local
--exec-prefix=EPREFIX	Install architecture-dependent files in EPREFIX; default to PREFIX
--bindir=DIR	User executables; default to $EPREFIX/bin
--libdir=DIR	Object code libraries; default to $EPREFIX/lib
--includedir=DIR	C header files; default to $PREFIX/include
--datarootdir=DIR	Read-only architecture-independent data root; default to $PREFIX/share
--datadir=DIR	Read-only architecture-independent data; default to $DATAROOTDIR
--docdir=DIR	Documentation root; default to $DATAROOTDIR/doc/linux-ums
--htmldir=DIR	Html documentation; default to $DOCDIR
--pdfdir=DIR	Pdf documentation; default to $DOCDIR
--build=BUILD	Configure for building on BUILD; guessed by default
--host=HOST	Cross-compile to build programs to run on HOST; default to $BUILD
--enable-silent-rules	Less verbose build output (undo: make V=1)
--disable-silent-rules	Verbose build output (undo: make V=0)

2.3.2 Variables

Some influential environment variables:

CC	C compiler command
CFLAGS	C compiler flags
LDFLAGS	Linker flags
LIBS	Libraries to pass to the linker
CPPFLAGS	C preprocessor flags
LT_SYS_LIBRARY_PATH	User-defined run-time library search path.
CPP	C preprocessor

2.3.3 Targets

`check`	Run code check
`html`	Build the html documentation
`pdf`	Build the pdf documentation
`install`	Install into the system
`installcheck`	Install tests into the system
`uninstall`	Uninstall from the system
`clean`	Delete all files in the current directory that are normally created by building the program
`distclean`	Delete all files in the current directory that are created by configuring or building the program

2.4 Minimal installation steps

```
$ ./configure
$ make
$ make check
$ sudo make install
$ sudo ldconfig
$ sudo modprobe ums
```

CHAPTER THREE

USERSPACE LIBRARY DOCUMENTATION

The user-space API manuals contain information about how to develop an application using UMS library.

3.1 Userspace library

3.1.1 UMS Scheduler

An application's UMS scheduler is responsible for creating, managing, and deleting UMS threads and determining which UMS thread to run. An application's scheduler performs the following tasks:

1. Creates one UMS scheduler thread for each processor on which the application will run UMS worker threads.
2. Creates UMS worker threads to perform the work of the application.
3. Maintains its own ready-thread queue of worker threads that are ready to run, and selects threads to run based on the application's scheduling policies.
4. Creates and monitors one or more completion lists where the system queues threads after they finish processing in the kernel. These include newly created worker threads and threads that had yielded.
5. Provides a scheduler entry point function to handles notifications from the system. The system calls the entry point function when a scheduler thread is created, when a worker thread explicitly yields control, or when a worker thread terminates.
6. Performs cleanup tasks for worker threads that have finished running.
7. Performs an orderly shutdown of the scheduler when requested by the application.

3.1.2 UMS Scheduler Thread

A UMS scheduler thread is an ordinary thread that has converted itself to UMS by calling the `enter_ums_scheduling_mode()` function. The system scheduler determines when the UMS scheduler thread runs based on its priority relative to other ready threads. The processor on which the scheduler thread runs is influenced by the thread's affinity, same as for non-UMS threads.

The caller of `enter_ums_scheduling_mode()` specifies a completion list and a `ums_scheduler_entry_point_t` entry point function to associate with the UMS scheduler thread. The system calls the specified entry point function when it is finished converting the calling thread to UMS. The scheduler entry point function is responsible for determining the appropriate next action for the specified thread.

An application might create one UMS scheduler thread for each processor that will be used to run UMS threads. The application might also set the affinity of each UMS scheduler thread for a specific logical processor, which tends to exclude unrelated threads from running on that processor, effectively reserving it for that scheduler thread. Be aware that setting thread affinity in this way can affect overall system performance by starving other processes that may be running on the system.

3.1.3 UMS Scheduler Entry Point Function

An application's scheduler entry point function is implemented as a *ums_scheduler_entry_point_t* function. The system calls the application's scheduler entry point function at the following times:

- When a non-UMS thread is converted to a UMS scheduler thread by calling *enter_ums_scheduling_mode()*.
- When a UMS worker thread calls *ums_thread_yield()*.
- When a UMS worker thread terminates.

The *ums_reason_t* parameter of the *ums_scheduler_entry_point_t* function specifies the reason that the entry point function was called. If the entry point function was called because a new UMS scheduler thread was created, the `scheduler_param` parameter contains data specified by the caller of *enter_ums_scheduling_mode()*. If the entry point function was called because a UMS worker thread yielded, the `scheduler_param` parameter contains data specified by the caller of *ums_thread_yield()*. If the entry point function was called because a UMS worker thread terminated, the `scheduler_param` parameter is NULL.

The scheduler entry point function is responsible for determining the appropriate next action for the specified thread.

When the scheduler entry point function is called, the application's scheduler should attempt to retrieve all of the items in its associated completion list by calling the *dequeue_ums_completion_list_items()* function. This function retrieves a list of UMS thread contexts that have finished processing in the kernel and are ready to run. The application's scheduler should not run UMS threads directly from this list because this can cause unpredictable behavior in the application. Instead, the scheduler should retrieve all UMS thread contexts by calling the *get_next_ums_list_item()* function once for each context, insert the UMS thread contexts in the scheduler's ready thread queue, and only then run UMS threads from the ready thread queue.

3.1.4 UMS Worker Threads and Completion Lists

A UMS worker thread is created by calling *ums_pthread_create()* and specifying a completion list.

A completion list is created by calling the *create_ums_completion_list()* function. A completion list receives UMS worker threads that have completed execution in the kernel and are ready to run. Only the system can enqueue worker threads to a completion list. New UMS worker threads are automatically enqueued to the completion list specified when the threads were created.

Each UMS scheduler thread is associated with a single completion list. However, the same completion list can be associated with any number of UMS scheduler threads, and a scheduler thread can retrieve UMS contexts from any completion list for which it has a pointer.

3.1.5 UMS Thread Execution

A newly created UMS worker thread is queued to the specified completion list and does not begin running until the application's UMS scheduler selects it to run. This differs from non-UMS threads, which the system scheduler automatically schedules to run.

The scheduler runs a worker thread by calling `execute_ums_thread()` with the worker thread's UMS context. A UMS worker thread runs until it yields by calling the `ums_thread_yield()` function or terminates.

3.2 API

3.2.1 Usage

In order to use the UMS library include its header:

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

3.2.2 Definitions and Data types

3.2.2.1 Defines

UMS_SCHEDULER_STARTUP
UMS scheduling proc activation due to scheduler thread startup

UMS_SCHEDULER_THREAD_YIELD
UMS scheduling proc activation due to worker thread yield

UMS_SCHEDULER_THREAD_END
UMS scheduling proc activation due to worker thread termination

3.2.2.2 Enums

enum **ums_reason_e**
UMS scheduling proc activation reason.

Values:

enumerator **UMS_SCHEDULER_STARTUP**

enumerator **UMS_SCHEDULER_THREAD_YIELD**

enumerator **UMS_SCHEDULER_THREAD_END**

3.2.2.3 Structs

struct **ums_scheduler_startup_info_s**

Specifies attributes for a user-mode scheduling (UMS) scheduler thread. The *enter_ums_scheduling_mode()* function uses this structure.

Public Members

ums_completion_list_t **completion_list**

An UMS completion list to associate with the calling thread.

ums_scheduler_entry_point_t **ums_scheduler_entry_point**

An application-defined *ums_scheduler_entry_point_t* entry point function. The system calls this function when the calling thread has been converted to UMS and is ready to run UMS worker threads. Subsequently, it calls this function when a UMS worker thread running on the calling thread yields or terminates.

void ***scheduler_param**

An application-defined parameter to pass to the specified *ums_scheduler_entry_point_t* function.

struct **ums_attr_s**

Specifies attributes for a user-mode scheduling (UMS) worker thread.

Public Members

ums_completion_list_t **completion_list**

An UMS completion list to associate with the worker thread. The newly created worker thread is queued to the specified completion list.

pthread_attr_t ***pthread_attr**

A pointer to a pthread attributes to configure the worker thread.

3.2.2.4 Unions

union **ums_activation_u**

#include <umsdefs.h> UMS scheduling proc activation.

Public Members

ums_context_t **context**
UMS worker thread context

3.2.2.5 Typedefs

typedef pid_t **ums_context_t**
UMS worker thread context.

typedef int **ums_completion_list_t**
UMS completion list.

typedef union *ums_activation_u* **ums_activation_t**
UMS scheduling proc activation.

typedef enum *ums_reason_e* **ums_reason_t**
UMS scheduling proc activation reason.

typedef void (***ums_scheduler_entry_point_t**)(*ums_reason_t* reason, *ums_activation_t* *activation, void *scheduler_param)
The application-defined user-mode scheduling (UMS) scheduler entry point function associated with a UMS completion list.

Parameters

- **reason** – **[in]** The reason the scheduler entry point is being called.
- **activation** – **[in]** If the `reason` parameter is *UMS_SCHEDULER_STARTUP* or *UMS_SCHEDULER_THREAD_END*, this parameter is NULL. If the `reason` parameter is *UMS_SCHEDULER_THREAD_YIELD*, this parameter contains the UMS thread context of the UMS worker thread that yielded.
- **scheduler_param** – **[in]** If the `reason` parameter is *UMS_SCHEDULER_STARTUP*, this parameter is the *ums_scheduler_startup_info_t::scheduler_param* member of the *ums_scheduler_startup_info_t* structure passed to the *enter_ums_scheduling_mode()* function that triggered the entry point call. If the `reason` parameter is *UMS_SCHEDULER_THREAD_YIELD* this parameter is the `scheduler_param` parameter passed to the *ums_thread_yield()* function that triggered the entry point call.

typedef struct *ums_scheduler_startup_info_s* **ums_scheduler_startup_info_t**
Specifies attributes for a user-mode scheduling (UMS) scheduler thread. The *enter_ums_scheduling_mode()* function uses this structure.

typedef struct *ums_attr_s* **ums_attr_t**
Specifies attributes for a user-mode scheduling (UMS) worker thread.

3.2.3 Functions

int **create_ums_completion_list**(*ums_completion_list_t* *completion_list)

Create UMS completion list.

A completion list is associated with a UMS scheduler thread when the *enter_ums_scheduling_mode()* function is called to create the scheduler thread. The system queues newly created UMS worker threads to the completion list.

When an application's *ums_scheduler_entry_point_t* entry point function is called, the application's scheduler should retrieve items from the completion list by calling *dequeue_ums_completion_list_items()*.

When a completion list is no longer needed, use the *delete_ums_completion_list()* to release the list. The list must be empty before it can be released.

Parameters

- `completion_list` – **[out]** pointer to an empty UMS completion list.

Returns 0 in case of success, -1 otherwise (with `errno` setted accordingly).

int **ums_pthread_create**(pthread_t *thread, *ums_attr_t* *ums_attr, void *(*func)(void*), void *args)

Create UMS worker thread.

Parameters

- `thread` – **[out]** pointer to an empty pthread.
- `ums_attr` – **[in]** pointer to a UMS worker attribute.
- `func` – **[in]** UMS worker routine.
- `args` – **[in]** parameter to pass to the specified worker routine.

Returns 0 in case of success, -1 otherwise (with `errno` setted accordingly).

int **enter_ums_scheduling_mode**(*ums_scheduler_startup_info_t* *scheduler_startup_info)

Converts the calling thread into a user-mode scheduling (UMS) scheduler thread.

An application's UMS scheduler creates one UMS scheduler thread for each processor that will be used to run UMS threads. The scheduler typically sets the affinity of the scheduler thread for a single processor, effectively reserving the processor for the use of that scheduler thread.

When a UMS scheduler thread is created, the system calls the *ums_scheduler_entry_point_t* entry point function specified with the *enter_ums_scheduling_mode()* function call. The application's scheduler is responsible for finishing any application-specific initialization of the scheduler thread and selecting a UMS worker thread to run.

The application's scheduler selects a UMS worker thread to run by calling *execute_ums_thread()* with the worker thread's UMS thread context. The worker thread runs until it yields control by calling *ums_thread_yield()* or terminates. The scheduler thread is then available to run another worker thread.

A scheduler thread should continue to run until all of its worker threads reach a natural stopping point: that is, all worker threads have yielded or terminated.

Parameters

- **scheduler_startup_info** – **[in]** A pointer to a structure that specifies UMS attributes for the thread, including a completion list and a *ums_scheduler_entry_point_t* entry point function.

Returns 0 in case of success, -1 otherwise (with `errno` setted accordingly).

int **dequeue_ums_completion_list_items**(*ums_completion_list_t* completion_list, *ums_context_t* *ums_thread_list)

Retrieves user-mode scheduling (UMS) worker threads from the specified UMS completion list.

The system queues a UMS worker thread to a completion list when the worker thread is created or when it yields. The *dequeue_ums_completion_list_items()* function retrieves a pointer to a list of all thread contexts in the specified completion list. The *get_next_ums_list_item()* function can be used to pop UMS thread contexts off the list into the scheduler's own ready thread queue. The scheduler is responsible for selecting threads to run based on priorities chosen by the application.

Do not run UMS threads directly from the list provided by *dequeue_ums_completion_list_items()*, or run a thread transferred from the list to the ready thread queue before the list is completely empty. This can cause unpredictable behavior in the application.

If more than one caller attempts to retrieve threads from a shared completion list, only the first caller retrieves the threads. For subsequent callers, the *dequeue_ums_completion_list_items()* function blocks until any UMS worker thread are queued to the completion list.

Parameters

- **completion_list** – **[in]** A pointer to the completion list from which to retrieve worker threads.
- **ums_thread_list** – **[out]** A pointer to a *ums_context_t* variable. On output, this parameter receives a pointer to the first UMS thread context in a list of UMS thread contexts.

Returns 0 in case of success, -1 otherwise (with `errno` setted accordingly).

ums_context_t **get_next_ums_list_item**(*ums_context_t* context)

Returns the next user-mode scheduling (UMS) thread context in a list of thread contexts.

Parameters

- **context** – **[in]** A UMS context in a list of thread contexts. This list is retrieved by the *dequeue_ums_completion_list_items()* function.

Returns If the function succeeds, it returns the next thread context in the list. If there is no thread context after the context specified by the `context` parameter, the function returns 0. If the function fails, the return value is -1 (with `errno` setted accordingly).

int **execute_ums_thread**(*ums_context_t* context)

Runs the specified UMS worker thread.

The *execute_ums_thread()* runs the specified UMS worker thread until it yields by calling the *ums_thread_yield()* function or terminates.

When a worker thread yields or terminates the system calls the scheduler thread's *ums_scheduler_entry_point_t* entry point function.

Parameters

- **context** – **[in]** The UMS thread context of the worker thread to run.

Returns 0 in case of success, -1 otherwise (with `errno` setted accordingly).

int **`ums_thread_yield`**(void *scheduler_param)

Yields control to the user-mode scheduling (UMS) scheduler thread on which the calling UMS worker thread is running.

A UMS worker thread calls the *ums_thread_yield()* function to cooperatively yield control to the UMS scheduler thread on which the worker thread is running. If a UMS worker thread never calls *ums_thread_yield()*, the worker thread runs until it is terminated.

When control switches to the UMS scheduler thread, the system calls the associated scheduler entry point function with the reason *UMS_SCHEDULER_THREAD_YIELD* and the `scheduler_param` parameter specified by the worker thread in the *ums_thread_yield(scheduler_param)* call.

The application's scheduler is responsible for rescheduling the worker thread.

Parameters

- **`scheduler_param`** – **[in]** A parameter to pass to the scheduler thread's *ums_scheduler_entry_point_t* function.

Returns 0 in case of success, -1 otherwise (with `errno` setted accordingly).

int **`delete_ums_completion_list`**(*ums_completion_list_t* *completion_list)

Deletes the specified user-mode scheduling (UMS) completion list.

Parameters

- **`completion_list`** – **[in]** A pointer to the UMS completion list to be deleted. The *create_ums_completion_list()* function provides this pointer.

Returns 0 in case of success, -1 otherwise (with `errno` setted accordingly).

3.3 Examples

3.3.1 Simple Example

The following example shows how to use UMS library with one UMS scheduler per-CPU core with their own ready queue.

```c
// SPDX-License-Identifier: AGPL-3.0-only

#include "global.h"
#include "list.h"

#include <ums.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <pthread.h>

#ifdef HAVE_SCHED_H
#include <sched.h>
#endif
```

(continues on next page)

(continued from previous page)

```

struct context_list_node {
        ums_context_t context;
        struct list_head list;
};

struct ums_sched_rq {
        struct list_head head;
};

ums_completion_list_t comp_list;
__thread struct ums_sched_rq rq;

static struct context_list_node *get_next_context(void)
{
        ums_context_t context;
        struct context_list_node *node;

        if (!list_empty(&rq.head)) {
                node = list_first_entry(&rq.head,
                                        struct context_list_node,
                                        list);
                list_del(&node->list);
                return node;
        }

        while (dequeue_ums_completion_list_items(comp_list, &context)) {
                if (errno == EINTR)
                        continue;
                else
                        return NULL;
        }

        node = malloc(sizeof(*node));
        if (!node)
                return NULL;
        node->context = context;

        list_add_tail(&node->list, &rq.head);

        while ((context = get_next_ums_list_item(context)) > 0) {
                node = malloc(sizeof(*node));
                if (!node)
                        return NULL;
                node->context = context;

                list_add_tail(&node->list, &rq.head);
        }

```

(continues on next page)

(continued from previous page)

```
        node = list_first_entry(&rq.head,
                                struct context_list_node,
                                list);
        list_del(&node->list);

        return node;
}

static inline void execute_next_context(void)
{
        struct context_list_node *node;
        ums_context_t context;

        node = get_next_context();
        if (!node) {
                perror("get_next_context");
                return;
        }

        if (execute_ums_thread(node->context))
                perror("execute_ums_thread");

        free(node);
}

static void sched_entry_proc(ums_reason_t reason,
                             ums_activation_t *activation,
                             void *args)
{
        ums_context_t context;
        long worker_result;

        switch (reason) {
        case UMS_SCHEDULER_STARTUP:
                execute_next_context();
                break;
        case UMS_SCHEDULER_THREAD_YIELD:
                context = activation->context;
                worker_result = *((long *) args);

                printf("worker %d yielded with value %ld\n",
                       context,
                       worker_result);
                fflush(stdout);

                free(args);

                execute_next_context();
                break;
```

(continues on next page)

(continued from previous page)

```
        case UMS_SCHEDULER_THREAD_END:
                execute_next_context();
                break;
        default:
                break;
        }
}

static void *sched_pthread_proc(void *arg)
{
        ums_scheduler_startup_info_t sched_info;

        sched_info.completion_list = comp_list;
        sched_info.ums_scheduler_entry_point = sched_entry_proc;

        INIT_LIST_HEAD(&rq.head);

#if !HAVE_DECL_PTHREAD_ATTR_SETAFFINITY_NP && defined(HAVE_SCHED_SETAFFINITY)
        (void) sched_setaffinity(0, sizeof(cpu_set_t), arg);
        free(arg);
#endif

        if (enter_ums_scheduling_mode(&sched_info))
                perror("enter_ums_scheduling_mode");

        return NULL;
}

int initialize_ums_scheduling(pthread_t *sched_threads,
                              long nthreads)
{
        pthread_attr_t attr;
        cpu_set_t cpus, *cpus_arg = NULL;
        long i;

        if (pthread_attr_init(&attr))
                return -1;

        if (create_ums_completion_list(&comp_list))
                return -1;

        for (i = 0L; i < nthreads; i++) {
                CPU_ZERO(&cpus);
                CPU_SET(i, &cpus);
                if (pthread_attr_setdetachstate(&attr,
                                                PTHREAD_CREATE_DETACHED))
                        goto out;
#if HAVE_DECL_PTHREAD_ATTR_SETAFFINITY_NP
                if (pthread_attr_setaffinity_np(&attr,
```

(continues on next page)

(continued from previous page)

```
                                                        sizeof(cpu_set_t),
                                                        &cpus))
                                goto out;

#else /* !HAVE_DECL_PTHREAD_ATTR_SETAFFINITY_NP */
#ifdef HAVE_SCHED_SETAFFINITY
                        cpus_arg = malloc(sizeof(*cpus_arg));
                        if (!cpus_arg)
                                goto out;
                        *cpus_arg = cpus;
#endif
#endif /* !HAVE_DECL_PTHREAD_ATTR_SETAFFINITY_NP */
                        if (pthread_create(sched_threads + i,
                                           &attr,
                                           sched_pthread_proc,
                                           cpus_arg))
#if !HAVE_DECL_PTHREAD_ATTR_SETAFFINITY_NP && defined(HAVE_SCHED_SETAFFINITY)
                                goto sched_thread_create;
#else
                                goto out;
#endif
        }

        return 0;

sched_thread_create:
        free(cpus_arg);
out:
        delete_ums_completion_list(&comp_list);
        return -1;
}

int release_ums_scheduling(pthread_t *sched_threads,
                           long nthreads)
{
        return delete_ums_completion_list(&comp_list);
}

int create_ums_worker_thread(pthread_t *thread, void *(*func)(void *),
                             void *arg)
{
        ums_attr_t attr;

        attr.completion_list = comp_list;
        attr.pthread_attr = NULL;

        return ums_pthread_create(thread, &attr, func, arg);
}

```

(continues on next page)

(continued from previous page)

```
static void *worker_pthread_proc(void *arg)
{
	long *result;

	result = malloc(sizeof(*result));
	if (!result)
		return NULL;

	*result = (long) (intptr_t) arg;

	if (ums_thread_yield(result))
		perror("ums_thread_yield");

	return NULL;
}

int main(int argc, char **argv)
{
	long			nproc = sysconf(_SC_NPROCESSORS_ONLN);
	pthread_t		sched_threads[nproc];
	long			nworkers = 24L * nproc;
	pthread_t		workers[nworkers];
	long			i;

	if (initialize_ums_scheduling(sched_threads, nproc)) {
		perror("initialize_ums_scheduling");
		return 1;
	}

	for (i = 0L; i < nworkers; i++) {
		if (create_ums_worker_thread(workers + i,
					     worker_pthread_proc,
					     (void *) i))
			perror("create_ums_worker_thread");
	}

	for (i = 0L; i < nworkers; i++)
		pthread_join(workers[i], NULL);

	if (release_ums_scheduling(sched_threads, nproc)) {
		perror("release_ums_scheduling");
		return 1;
	}

	return 0;
}
```

3.4 Library internals

3.4.1 UMS library lifecycle

All UMS objects are private to each process that opens the UMS device. Indeed every process that opens the UMS device gets allocated a private pool of UMS schedulers, UMS workers and completion lists.

Every time that the library is loaded the UMS device is opened and a per-process file descriptor, named `UMS_FILENO`, is initialized. All subsequentes `ioctl` calls for interacting with the kernel module are performed upon that file descriptor. Every time that the library is unloaded the UMS device is closed, releasing all kernel resources that are associated with the process.

That initialization and deinitialization is implemented through constructor and destructor functions annotated respectively with `__attribute__((constructor))` and `__attribute__((destructor))` attributes.

In particular the `UMS_FILENO` is declared as

Listing 1: src/lib/src/private.h

```c
extern int UMS_FILENO;
```

and it is managed at

Listing 2: src/lib/src/hooks.c

```c
/**
 * @brief Open UMS file descriptor
 */
static void ums_fileno_open(void)
{
        UMS_FILENO = open("/dev/" UMS_DEV_NAME, O_RDONLY);
}

/**
 * @brief Close UMS file descriptor
 */
static void ums_fileno_close(void)
{
        close(UMS_FILENO);
}

/**
 * @brief Handler executed in the child process after fork processing␣
↪completes
 */
static void ums_atfork_child_handler(void)
{
        ums_fileno_close();
        ums_fileno_open();
}

```

(continues on next page)

(continued from previous page)

```
/**
 * @brief UMS library ctor
 *
 * This function initializes UMS library.
 */
__attribute__((constructor))
static void ums_init(void)
{
        ums_fileno_open();
        (void) pthread_atfork(NULL, NULL, ums_atfork_child_handler);
}

/**
 * @brief UMS library dtor
 *
 * This function deinitializes UMS library.
 */
__attribute__((destructor))
static void ums_exit(void)
{
        ums_fileno_close();
}
```

Due to file descriptors inheritance between parent and child processes, the `pthread_atfork()` function registers an handler that is executed in the child process after `fork()` processing completes; that handler is in charge of closing the `UMS_FILENO` and reopening it, resulting thus in two separate pools of UMS resources for the two processes.

3.4.2 UMS Scheduler

A UMS scheduler thread is a regular pthread that has entered the UMS scheduling mode.

3.4.2.1 Enter UMS scheduling mode

A regular pthread is converted into a UMS scheduler thread performing a `ioctl` call with *`IOCTL_ENTER_UMS`* request parameter; in particular the *`enter_ums_scheduling_mode()`* begins with:

Listing 3: src/lib/src/scheduler.c

```
struct enter_ums_mode_args enter_args = {
        .flags = ENTER_UMS_SCHED
};
struct ums_sched_event event;
ums_activation_t scheduler_activation;

if (!scheduler_startup_info) {
        errno = EFAULT;
        return -1;
```

(continues on next page)

(continued from previous page)

```
}

enter_args.ums_complist = scheduler_startup_info->completion_list;

if (enter_ums_mode(&enter_args))
        return -1;
```

where the enter_ums_mode is defined at

Listing 4: src/lib/src/private.h

```
static __always_inline int enter_ums_mode(struct enter_ums_mode_args *args)
{
        return ioctl(UMS_FILENO, IOCTL_ENTER_UMS, args);
}
```

After a pthread has entered UMS scheduling mode it starts an infinite loop waiting for UMS scheduler events. All scheduling activities are passed from kernel space through *ums_sched_event* events and then proxied to the UMS *ums_scheduler_entry_point_t*.

Listing 5: src/lib/src/scheduler.c

```
for (;;) {
        if (dequeue_ums_sched_event(&event)) {
                if (errno == EINTR) continue;
                else return -1;
        }

        // proxies event to ums scheduler entry point
        switch (event.type) {
        case SCHEDULER_STARTUP:
                scheduler_startup_info->ums_scheduler_entry_point(
                        UMS_SCHEDULER_STARTUP,
                        NULL,
                        scheduler_startup_info->scheduler_param
                );
                break;
        case THREAD_YIELD:
                scheduler_activation.context =
                                        event.yield_params.context;

                scheduler_startup_info->ums_scheduler_entry_point(
                        UMS_SCHEDULER_THREAD_YIELD,
                        &scheduler_activation,
                        event.yield_params.scheduler_params
                );
                break;
        case THREAD_TERMINATED:
                scheduler_activation.context =
                                        event.end_params.context;
```

(continues on next page)

(continued from previous page)

```
                scheduler_startup_info->ums_scheduler_entry_point(
                        UMS_SCHEDULER_THREAD_END,
                        &scheduler_activation,
                        NULL
                );
                break;
        default:
                break;
        }
}
```

3.4.2.2 Execute UMS worker thread context

The execution of a worker thread context is implemented as follows:

Listing 6: src/lib/src/scheduler.c

```
int execute_ums_thread(ums_context_t context)
{
        return ioctl(UMS_FILENO, IOCTL_EXEC_UMS_CTX, context);
}
```

3.4.3 UMS Worker

A UMS worker thread is a regular pthread that has entered the UMS working mode.

3.4.3.1 Enter UMS working mode

A UMS worker thread is created by calling the *ums_pthread_create()* function.

The function is implemented as follows:

Listing 7: src/lib/src/worker.c

```
int ums_pthread_create(pthread_t *thread, ums_attr_t *ums_attr,
                             void *(*func)(void *), void *args)
{
        worker_proc_args_t *ums_args;

        if (!ums_attr || !func) {
                errno = EFAULT;
                return -1;
        }

        ums_args = malloc(sizeof(*ums_args));
        if (!ums_args)
                return -1;
```

(continues on next page)

(continued from previous page)

```

        ums_args->completion_list = ums_attr->completion_list;
        ums_args->func = func;
        ums_args->args = args;

        return pthread_create(thread,
                              ums_attr->pthread_attr,
                              worker_wrapper_routine,
                              ums_args);
}
```

where `worker_proc_args_t` is defined as

Listing 8: src/lib/src/worker.c

```
typedef struct worker_proc_args_s {
        ums_completion_list_t completion_list;
        void *(*func)(void *);
        void *args;
} worker_proc_args_t;
```

The `worker_wrapper_routine` is the routine executed by the newly created pthread: it starts entering the UMS worker mode and then executing the user specified function.

In particular:

Listing 9: src/lib/src/worker.c

```
static pthread_key_t worker_key;
static pthread_once_t worker_key_once = PTHREAD_ONCE_INIT;

static inline int exit_ums_mode(void)
{
        return ioctl(UMS_FILENO, IOCTL_EXIT_UMS);
}

static void destroy_worker_key(void *args)
{
        free(args);
        (void) exit_ums_mode();
}

static void create_worker_key(void)
{
        (void) pthread_key_create(&worker_key, destroy_worker_key);
}

static void *worker_wrapper_routine(void *args)
{
        worker_proc_args_t *worker_args = args;

```

(continues on next page)

(continued from previous page)

```
        struct enter_ums_mode_args ums_args = {
                .flags = ENTER_UMS_WORK,
                .ums_complist = worker_args->completion_list
        };

        (void) pthread_once(&worker_key_once, create_worker_key);

        // enter ums mode and deschedule worker thread (suspend here)
        if (enter_ums_mode(&ums_args)) {
                free(args);
                return NULL;
        }

        (void) pthread_setspecific(worker_key, args);

        // worker thread is now active for scheduling
        return worker_args->func(worker_args->args);
}
```

The `worker_key` contains the unique `worker_wrapper_routine` argument specific for each worker thread. When the user specified function calls `pthread_exit()`, returns or is cancelled because of a `pthread_cancel()` request, the destructor routine `destroy_worker_key` is called releasing every resource associated with that UMS worker thread.

3.4.3.2 UMS worker yield

The yielding of a UMS worker thread is implemented as follows:

Listing 10: src/lib/src/worker.c

```
int ums_thread_yield(void *scheduler_param)
{
        return ioctl(UMS_FILENO, IOCTL_UMS_YIELD, scheduler_param);
}
```

3.4.4 UMS Completion List

The UMS completion list methods are simply implemented by performing `ioctl` calls as described at *uAPI Usage*.

CHAPTER

FOUR

KERNEL MODULE DOCUMENTATION

4.1 uAPI

The kernel module creates a miscellaneous character device located at `/dev/ums` with which the user application can interact through `ioctl` calls. All UMS objects are private to each process that opens the UMS device. Indeed every process that opens the UMS device gets allocated a private pool of UMS schedulers, UMS workers and completion lists.

4.1.1 Definitions and Data types

4.1.1.1 Defines

`UMS_DEV_NAME`
UMS device name

`ENTER_UMS_SCHED`
Enter UMS mode as UMS scheduler thread

`ENTER_UMS_WORK`
Enter UMS mode as UMS worker thread

`IOCTL_CREATE_UMS_CLIST`
Create UMS completion list IOCTL number

`IOCTL_ENTER_UMS`
Enter UMS mode IOCTL number

`IOCTL_UMS_SCHED_DQEVENT`
Dequeue UMS scheduler event IOCTL number

`IOCTL_DEQUEUE_UMS_CLIST`
Dequeue UMS context from completion list IOCTL number

`IOCTL_NEXT_UMS_CTX_LIST`
Get next completion list UMS context IOCTL number

IOCTL_EXEC_UMS_CTX
Execute UMS worker thread IOCTL number

IOCTL_UMS_YIELD
UMS worker thread yield IOCTL number

IOCTL_EXIT_UMS
Worker thread exit UMS mode IOCTL number

IOCTL_DELETE_UMS_CLIST
Delete UMS completion list IOCTL number

4.1.1.2 Enums

enum **ums_sched_event_type_e**
UMS scheduler event type

Values:

enumerator **SCHEDULER_STARTUP**
UMS scheduler startup event

enumerator **THREAD_YIELD**
UMS worker yielded event

enumerator **THREAD_TERMINATED**
UMS worker terminated event

4.1.1.3 Structs

struct **enter_ums_mode_args**
struct for enter UMS mode

Public Members

int **flags**
specify caller UMS mode

ums_comp_list_id_t **ums_complist**
the completion list to be associated with the caller

struct **ums_thread_yield_args**
UMS scheduler event associated to a UMS worker thread that yielded

Public Members

pid_t **context**
UMS context of the worker thread that yielded.

void ***scheduler_params**
parameter passed from the UMS worker thread.

struct **ums_thread_end_args**
UMS scheduler event associated to a UMS worker thread that terminated

Public Members

pid_t **context**
UMS context of the worker thread that terminated.

struct **ums_sched_event**
UMS scheduler event

Public Members

ums_sched_event_type_t **type**

struct *ums_thread_yield_args* **yield_params**

struct *ums_thread_end_args* **end_params**

union *ums_sched_event*.[anonymous] **[anonymous]**
UMS scheduler event parameters

struct **dequeue_ums_complist_args**
Dequeue UMS completion list args

Public Members

ums_comp_list_id_t **ums_complist**
the completion list from which dequeuying UMS contexts

pid_t **ums_context**
dequeued UMS context

struct **ums_next_context_list_args**
Get next UMS context list args

Public Members

pid_t **ums_context**
current UMS context in the dequeued UMS thread list

pid_t **ums_next_context**
next UMS context in the dequeued UMS thread list

4.1.1.4 Typedefs

typedef int **ums_comp_list_id_t**
UMS completion list id.

typedef enum *ums_sched_event_type_e* **ums_sched_event_type_t**
UMS scheduler event type

4.1.2 Usage

In order to use the UMS device include the header:

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

4.1.2.1 Open UMS device

```
int ums_dev_fd;

ums_dev_fd = open("/dev/" UMS_DEV_NAME, O_RDONLY);
```

4.1.2.2 Create UMS completion list

```
ums_comp_list_id_t comp_list;

ioctl(ums_dev_fd, IOCTL_CREATE_UMS_CLIST, &comp_list);
```

4.1.2.3 Enter UMS scheduler mode

```
struct enter_ums_mode_args enter_args = {
    .flags = ENTER_UMS_SCHED,
    .ums_complist = comp_list
};

ioctl(ums_dev_fd, IOCTL_ENTER_UMS, &enter_args);
```

4.1.2.4 Enter UMS worker mode

```
struct enter_ums_mode_args enter_args = {
    .flags = ENTER_UMS_WORK,
    .ums_complist = comp_list
};

ioctl(ums_dev_fd, IOCTL_ENTER_UMS, &enter_args);
```

4.1.2.5 Dequeue UMS scheduler event

```
struct ums_sched_event event;

ioctl(ums_dev_fd, IOCTL_UMS_SCHED_DQEVENT, &event);
```

4.1.2.6 Dequeue UMS completion list items

```
struct dequeue_ums_complist_args dequeue_args = {
    .ums_complist = comp_list
};
ums_context_t context;

ioctl(ums_dev_fd, IOCTL_DEQUEUE_UMS_CLIST, &dequeue_args);
context = dequeue_args.ums_context;
```

4.1.2.7 Get next UMS context list item

```
struct ums_next_context_list_args next_context_args = {
    .ums_context = context
};
ums_context_t next_context;

ioctl(ums_dev_fd, IOCTL_NEXT_UMS_CTX_LIST, &next_context_args);
next_context = next_context_args.ums_next_context;
```

4.1.2.8 Execute UMS worker context

```
ums_context_t context;

ioctl(ums_dev_fd, IOCTL_EXEC_UMS_CTX, context);
```

4.1.2.9 UMS worker yield

```
void *yield_param;

ioctl(ums_dev_fd, IOCTL_UMS_YIELD, yield_param);
```

4.1.2.10 Exit UMS worker mode

```
ioctl(ums_dev_fd, IOCTL_EXIT_UMS);
```

4.1.2.11 Delete UMS completion list

```
ums_comp_list_id_t comp_list;

ioctl(ums_dev_fd, IOCTL_DELETE_UMS_CLIST, comp_list);
```

4.2 Module internals

4.2.1 Module lifecycle

The linux kernel module implements all backend services for executing and yielding UMS worker threads.

Once loaded, the kernel module initializes its caches, it registers the UMS device and it initializes the UMS `procfs` directories:

Listing 1: src/module/src/ums_mod.c

```
static int __init ums_init(void)
{
        int retval;

        retval = ums_caches_init();
        if (retval)
                goto cache_init;

        retval = register_ums_device();
        if (retval)
                goto register_dev;

        retval = ums_proc_init();
        if (retval)
                goto proc_init;

        return 0;

```

(continues on next page)

(continued from previous page)

```c
proc_init:
        unregister_ums_device();
register_dev:
        ums_caches_destroy();
cache_init:
        return retval;
}
```

When the module is unloaded it destroies every resources associated with it:

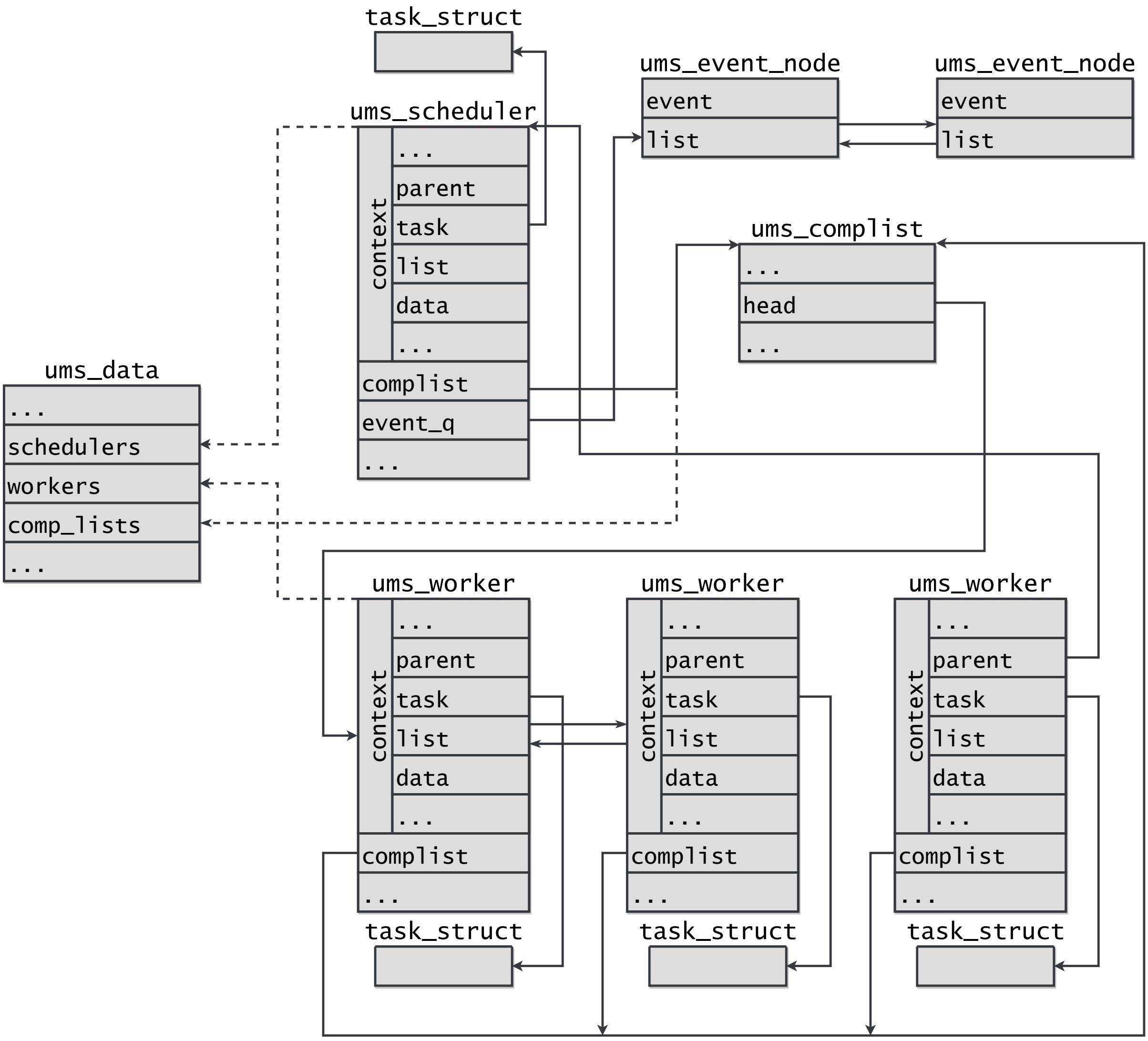
Listing 2: src/module/src/ums_mod.c

```c
static void __exit ums_exit(void)
{
        /* wait for all RCU callbacks to fire. */
        rcu_barrier();

        unregister_ums_device();
        ums_caches_destroy();
        ums_proc_destroy();
}
```

4.2.2 Module diagram

The relations between module data structures are depicted as follows:



4.2.3 Core structures and functionalities

4.2.3.1 Logging

```
#include <log.h>
```

pr_fmt(fmt)

Defaul module `printk` format.

4.2.3.2 ID allocation

```
#include <idr_l.h>
```

struct **idr_l**

IDR structure wrapper with internal concurrency handling.

Public Members

struct *idr* **idr**

IDR_L_INIT(idr_l)
Initialise a IDR.

Parameters

- **idr_l** – pointer to the IDR

IDR_L_ALLOC(idr_l, ptr, flags)
Allocate a new ID.

Parameters

- **idr_l** – pointer to the IDR
- **ptr** – **[in]** pointer to be associated with the new ID
- **flags** – **[in]** memory allocation flags

Returns

- the newly allocated ID
- -ENOMEM - No memory available
- -ENOSPC - No free IDs could be found

IDR_L_FIND(idr_l, id)
Return pointer for given ID.

Looks up the pointer associated with this ID. A NULL pointer may indicate that `id` is not allocated or that the NULL pointer was associated with this ID.

This function can be called under `rcu_read_lock()`.

Parameters

- **idr_l** – pointer to the IDR
- **id** – **[in]** pointer ID

Returns the pointer associated with this ID

IDR_L_FOR_EACH(idr_l, func, data)
Iterate through all stored pointers.

The callback function will be called for each entry in *idr_l*, passing the ID, the entry and `data`.

If `func` returns anything other than 0, the iteration stops and that value is returned from this function.

IDR_L_FOR_EACH() can be called concurrently with *IDR_L_ALLOC()* and *IDR_L_REMOVE()* if protected by RCU. Newly added entries may not be seen and deleted entries may be seen, but adding and removing entries will not cause other entries to be skipped, nor spurious ones to be seen.

Parameters

- **idr_l** – pointer to the IDR

- **func** – **[in]** function to be called for each pointer
- **data** – **[in]** data passed to callback function

`IDR_L_REMOVE`(idr_l, id)

Remove an ID from the IDR.

Removes this ID from the IDR. If the ID was not previously in the IDR, this function returns NULL.

Parameters

- **idr_l** – pointer to the IDR
- **id** – **[in]** pointer ID

Returns the pointer formerly associated with this ID

`IDR_L_DESTROY`(idr_l)

Release all internal memory from an IDR.

After this function is called, the IDR is empty, and may be reused or the data structure containing it may be freed.

A typical clean-up sequence for objects stored in an idr tree will use *IDR_L_FOR_EACH()* to free all objects, if necessary, then *IDR_L_DESTROY()* to free the memory used to keep track of those objects.

Parameters

- **idr_l** – pointer to the IDR

4.2.4 UMS Device

The UMS device is registered once the kernel module is loaded. It supports three operations: open, ioctl and release.

4.2.4.1 Open

When the UMS device is opened all required resources are allocated and initialized. In particular the private data of the device points to a local `ums_data` struct.

4.2.4.2 Release

When the UMS device is closed all UMS resources that are still present in the `ums_data` struct are released.

4.2.4.3 Structs

struct **ums_data**

UMS device private data.

Public Members

struct *idr_l* **comp_lists**

Pool of UMS completion lists

struct rhashtable **schedulers**

Pool of UMS schedulers

struct rhashtable **workers**

Pool of UMS workers

struct *ums_proc_dirs* **dirs**

Procfs base dirs

4.2.4.4 Functions

inline int **ums_caches_init**(void)

Initialize UMS device caches.

Returns 0 in case of success, -ERRNO otherwise

inline void **ums_caches_destroy**(void)

Destroy UMS device caches.

inline int **register_ums_device**(void)

Register UMS miscellaneous device.

Returns 0 in case of success, -ERRNO otherwise.

inline void **unregister_ums_device**(void)

Unregister UMS miscellaneous device.

Returns 0 in case of success, -ERRNO otherwise.

inline int **enter_ums_mode**(struct *ums_data* *data, struct *enter_ums_mode_args* *args)

Enter UMS mode.

Enter UMS mode as a UMS scheduler or a UMS worker depending on `args::flags`.

Context: Process context. May sleep.

Parameters

- **data** – **[in]** pointer to the UMS data
- **args** – **[in]** userspace pointer for args

Returns

- 0 - OK

- -EACCESS - Bad userspace pointer
- -ENOMEM - No memory available
- -EINVAL - Invalid args

4.2.5 UMS Context

A UMS context represents the state of a UMS thread.

Every UMS context is used for indexing its UMS thread owner by means of the `ums_context.pid` key.

Both a *UMS scheduler* and a *UMS worker* contains their own UMS context.

The UMS context implements all facilities for switching from UMS schedulers and UMS workers and viceversa.

4.2.5.1 Defines

`CONTEXT_RUNNING`
UMS context running state

`CONTEXT_IDLE`
UMS context idle state

`__set_context_state`(ctx, state_value)
Set context state.

Context: Any context.

Parameters

- **`ctx`** – **[in]** pointer to a UMS context
- **`state_value`** – **[in]** new state value

`set_context_state`(ctx, state_value)
Set context state and perform a write memory barrier.

Context: Any context.

Parameters

- **`ctx`** – **[in]** pointer to a UMS context
- **`state_value`** – **[in]** new state value

4.2.5.2 Structs

struct **`ums_context`**
UMS context structure

Public Members

unsigned int **`state`**
context state

struct task_struct ***`task`**
`task_struct` associated with the UMS context

pid_t **`pid`**
context key id

struct *ums_context* ***`parent`**
parent context

struct rhash_head **`node`**
context hashtable node

atomic_t **`switches`**
context switches counter

struct list_head **`list`**
head to context list forming a completion list

struct *rcu_head* **`rcu_head`**
rcu head

struct *ums_data* ***`data`**
pointer to UMS device private data where this context belongs to

4.2.5.3 Functions

void **`ums_context_init`**(struct *ums_context* *context)
Initialize a UMS context.

Set the context task as `current` and increment its reference counter.

The initialized context state is setted to *CONTEXT_RUNNING*.

Context: Any context.

Parameters

- **`context`** – **[in]** pointer to a UMS context void **`ums_context_deinit`**(struct *ums_context* *context)

Deinitialize a UMS context.

Decrement context task reference counter.

Context: Any context.

Parameters

- **`context`** – **[in]** pointer to a UMS context

static inline pid_t **`current_context_pid`**(void)

Retrieve current UMS context pid.

Context: Any context.

Returns The virtual pid of the current UMS context.

static inline void **`prepare_suspend_context`**(struct *ums_context* *context)

Initialize suspending UMS context.

Change context state to *CONTEXT_IDLE*

Context: Any context.

Parameters

- **`context`** – **[in]** pointer to a UMS context

static inline void **`wake_up_context`**(struct *ums_context* *context)

Wake up an idle UMS context.

Change context state to *CONTEXT_RUNNING* and wake up the context task.

Context: Any context.

Parameters

- **`context`** – **[in]** pointer to a UMS context

void **`prepare_switch_context`**(struct *ums_context* *from, struct *ums_context* *to)

Initialize a UMS context switch.

Assign the `from` context to the `to->parent`; suspend the `from` context and wake up the `to` context.

Context: Any context.

Parameters

- **`from`** – **[in]** pointer to the UMS context from which start the switch
- **`to`** – **[in]** pointer to the UMS context to which end the switch

4.2.6 UMS Completion List

4.2.6.1 Overview

The life cycle of a UMS completion list is bounded to the process that opens the UMS device and it is represented by the *ums_complist* structure. Once created it is stored inside the pool of completion lists pointed by *ums_data.comp_lists*. The completion list pool is based on a *idr_l* structure where a new ID is allocated for every completion list and it is then used for retrieving a completion list.

The memory allocated for the *ums_complist* structure is managed through its *ums_complist.refcount* and *ums_complist.rhead*: everyone who needs a reference to the completion list has to increment its reference counter and decrement it when it has finished using the completion list; when the reference counter of the completion list reaches zero the memory allocated to it is released through a `call_rcu` directive.

The list of UMS worker's contexts associated with a particular UMS completion list is pointed by *ums_complist.head*; when an UMS scheduler dequeues the UMS workers from a completion list its *ums_complist.head* is transferred to the first UMS worker's context of the list who becomes the new handler for retrieving the next UMS worker's contexts.

4.2.6.2 Defines

COMPLIST_ADD_HEAD
Add a UMS context to the head of the UMS completion list

COMPLIST_ADD_TAIL
Add a UMS context to the tail of the UMS completion list

4.2.6.3 Structs

struct **ums_complist**
UMS completion list structure

Public Members

ums_comp_list_id_t **id**
UMS completion list id

struct list_head **head**
list head to first UMS context

spinlock_t **lock**

wait_queue_head_t **wait_q**
wait queue for dequeuing UMS contexts

struct kref **refcount**
Reference counter

struct rcu_head **rhead**
rcu head

struct *ums_data* ***data**
pointer to UMS device private data where this context belongs to

4.2.6.4 Functions

int **create_ums_complist**(struct *ums_data* *data, *ums_comp_list_id_t* *id)
Create a UMS completion list.

Allocates a *ums_complist* and initialize all its data structures.

The *ums_complist::refcount* counter is incremented and its *ums_complist::id* is passed to the user.

Context: Process context. May sleep.

Parameters

- **data** – **[in]** pointer to the UMS data
- **id** – **[in]** userspace pointer to the completion list id

Returns

- 0 - OK
- -ENOMEM - No memory available
- -EACCESS - Bad userspace pointer

void **ums_completion_list_add**(struct *ums_complist* *complist, struct *ums_context* *context, unsigned int flags)
Add a UMS context to a UMS completion list.

Context: Process context. Takes and releases `complist->lock`.

Parameters

- **complist** – **[in]** pointer to the UMS completion list
- **context** – **[in]** pointer to the UMS worker context
- **flags** – **[in]** flags specifying how to add the context (can be one from *COMPLIST_ADD_HEAD* or *COMPLIST_ADD_TAIL*)

int **ums_complist_dqcontext**(struct *ums_data* *data, struct *dequeue_ums_complist_args* *args)
Retrieve a UMS context from a UMS completion list.

Context: Process context. May sleep. Takes and releases the RCU lock. Takes and releases `complist->lock`.

Parameters

- **data** – **[in]** pointer to the UMS private data
- **args** – **[in]** Userspace pointer for args

Returns

- 0 - OK

- -EACCESS - Bad userspace pointer
- -EINVAL - Bad completion list id
- -ESRCH - Bad UMS calling thread
- -EINTR - Interrupted call

int **ums_complist_next_context**(struct *ums_data* *data, struct *ums_next_context_list_args* *args)
Retrieve the next UMS context from a UMS thread list.

Context: Process context. May sleep. Takes and releases the RCU lock.

Parameters

- **data** – **[in]** pointer to the UMS private data
- **args** – **[in]** Userspace pointer for args

Returns

- 0 - OK
- -EACCESS - Bad userspace pointer
- -ESRCH - Bad UMS context

static struct *ums_complist* ***get_ums_complist**(struct *ums_complist* *c)
Increment the UMS completion list reference counter.

Context: Any context.

Parameters

- **c** – **[in]** pointer to the UMS completion list

Returns the UMS completion list

int **put_ums_complist**(struct *ums_complist* *complist)
Decrement the UMS completion list reference counter and destroy it if the counter reaches zero.

Context: Any context.

Parameters

- **complist** – **[in]** pointer to the UMS completion list

Returns 1 if the completion list is destroyed, 0 otherwise.

int **ums_complist_delete**(struct *ums_data* *data, *ums_comp_list_id_t* complist_id)
Delete the UMS completion list.

Decrement the UMS completion list reference counter and destroy it if the counter reaches zero.

Context: Any context. Takes and releases the RCU lock.

Parameters

- **data** – **[in]** pointer to the UMS private data
- **complist_id** – **[in]** id of the UMS completion list

Returns

- 0 - OK
- -EINVAL - Bad UMS completion list id

4.2.7 UMS Scheduler

4.2.7.1 Overview

The life cycle of a UMS scheduler is bounded to the process that opens the UMS device and it is represented by the *ums_scheduler* structure. Once created, it is stored inside the pool of schedulers pointed by *ums_data.schedulers*. The kernel `rhashtable` structure has been choosen for representing the schedulers pool since the number of UMS schedulers that are going to be created isn't known in advance. In this way the size of the hashtable will be automatically adjusted providing better performances. Every UMS scheduler is indexed inside the pool by its *ums_scheduler.context*.

UMS scheduling activities are notified to userland process by means of *IOCTL_UMS_SCHED_DQEVENT* `ioctl` calls. After a userland thread converts itself to UMS scheduling thread a *ums_event_node* of type *ums_sched_event_type_e.SCHEDULER_STARTUP* is posted to the newly created scheduler's *ums_scheduler.event_q*; when an UMS worker thread yields or terminates a *ums_event_node* of type *ums_sched_event_type_e.THREAD_YIELD* or *ums_sched_event_type_e.THREAD_TERMINATED* is posted to the scheduler's event queue.

Every *ums_event_node* is allocated by a dedicated slab cache in order to speed up allocation time. The dedicated slab cache is created when the UMS module is loaded by calling *ums_scheduling_cache_create()* and destroied when the module is unloaded by calling *ums_scheduling_cache_destroy()*.

When a UMS scheduler is created it is registered to the dedicated UMS procfs at `/proc/ums/<pid>/schedulers/<scheduler-pid>`, where `<pid>` is the PID of the process that opens the UMS device and `<scheduler-pid>` is the PID of the UMS scheduler. Once a UMS scheduler terminates it is unregistered from the UMS procfs.

For every UMS worker that is dequeued by a UMS scheduler from its UMS completion list a symlink is created inside `/proc/ums/<pid>/schedulers/<scheduler-pid>/workers` folder and it is subsequently deleted when the UMS scheduler executes one of them. In this way at any time the `/proc/ums/<pid>/schedulers/<scheduler-pid>/workers` folder represents the list of UMS workers that an UMS scheduler is owning.

4.2.7.2 Defines

EVENT_ADD_HEAD
Add a UMS scheduling event to the head of the event queue

EVENT_ADD_TAIL
Add a UMS scheduling event to the tail of the event queue

4.2.7.3 Structs

struct **ums_scheduler**
UMS scheduler struct

Public Members

struct *ums_context* **context**
scheduler context

struct *ums_complist* ***complist**
scheduler completion list

struct list_head **event_q**
scheduler event queue list

spinlock_t **lock**
scheduler spinlock

wait_queue_head_t **sched_wait_q**
scheduler wait queue

struct *ums_scheduler_proc_dirs* **dirs**
scheduler procfs dirs

struct **ums_event_node**
UMS scheduler event node.

Public Members

struct *ums_sched_event* **event**

struct list_head **list**

4.2.7.4 Functions

int **ums_scheduling_cache_create**(void)
Create a slab cache for allocating UMS scheduling events.

Returns

- 0 - OK
- -ENOMEM - No memory available

void **ums_scheduling_cache_destroy**(void)
Destroy the slab cache for UMS scheduling events.

int **enter_ums_scheduler_mode**(struct *ums_data* *data, struct *enter_ums_mode_args* *args)

Enter UMS scheduling mode.

Creates a UMS scheduler and send a *SCHEDULER_STARTUP* event to it.

Context: Process context. May sleep. Takes and releases the RCU lock.

Parameters

- **data** – **[in]** pointer to the UMS data
- **args** – **[in]** userspace pointer for args

Returns

- 0 - OK
- -ENOMEM - No memory available
- -EINVAL - Invalid UMS completion list arg

struct *ums_event_node* ***alloc_ums_event**(void)

Allocate an UMS scheduling event from the dedicated slab cache.

Returns the UMS scheduling event

void **free_ums_event**(struct *ums_event_node* *event)

Deallocate the UMS scheduling event.

void **enqueue_ums_sched_event**(struct *ums_scheduler* *scheduler, struct *ums_event_node* *event, unsigned int flags)

Enqueue the UMS scheduling event to the UMS scheduler queue.

Context: Process context. Takes and releases `scheduler->lock`

Parameters

- **scheduler** – **[in]** pointer to the UMS scheduler
- **event** – **[in]** pointer to the UMS scheduling event
- **flags** – **[in]** flags specifying how to add the event (can be one from *EVENT_ADD_HEAD* or *EVENT_ADD_TAIL*)

void **ums_scheduler_destroy**(struct *ums_scheduler* *sched)

Destroy the UMS scheduler and every UMS scheduling event associated with it.

Context: Process context. Takes and releases `scheduler->lock`

Parameters

- **sched** – **[in]** pointer to the UMS scheduler

int **exec_ums_context**(struct *ums_data* *data, pid_t worker_pid)

Execute a UMS worker context.

Suspend the UMS scheduler switching its context with the UMS worker's one.

Context: Process context. May sleep. Takes and releases the RCU lock.

Parameters

- **data** – **[in]** pointer to the UMS data
- **worker_pid** – **[in]** pid of the UMS worker

Returns

- 0 - OK
- -ESRCH - Bad worker pid or bad UMS calling thread

4.2.8 UMS Worker

4.2.8.1 Overview

The life cycle of a UMS worker is bounded to the process that opens the UMS device and it is represented by the *ums_worker* structure. Once created, it is stored inside the pool of workers pointed by *ums_data.workers*. The kernel `rhashtable` structure has been choosen for representing the workers pool since the number of UMS workers that are going to be created isn't known in advance. In this way the size of the hashtable will be automatically adjusted providing better performances. Every UMS worker is indexed inside the pool by its *ums_worker.context*.

When a UMS worker is created it is registered to the dedicated UMS procfs at `/proc/ums/<pid>/workers/<worker-pid>`, where `<pid>` is the PID of the process that opens the UMS device and `<worker-pid>` is the PID of the UMS worker. Once a UMS worker terminates it is unregistered from the UMS procfs.

4.2.8.2 Structs

struct **ums_worker**
UMS worker struct

Public Members

struct *ums_context* **context**
worker context

struct *ums_complist* ***complist**
worker completion list

struct *ums_worker_proc_dirs* **dirs**
worker procfs dirs

4.2.8.3 Functions

int **enter_ums_worker_mode**(struct *ums_data* *data, struct *enter_ums_mode_args* *args)
Enter UMS worker mode.

Creates a UMS worker, add its UMS context to the UMS completion list and suspend the UMS worker.

Context: Process context. May sleep. Takes and releases the RCU lock.

Parameters

- **data** – **[in]** pointer to the UMS data

- **args** – **[in]** userspace pointer for args

Returns

- 0 - OK
- -ENOMEM - No memory available
- -EINVAL - Invalid UMS completion list arg

int **ums_worker_yield**(struct *ums_data* *data, void *args)

Yield UMS worker.

Add the UMS worker context to the UMS completion list, enqueue a UMS scheduling event of type *THREAD_YIELD* to the parent UMS scheduler and switch the UMS context with the parent one.

Context: Process context. May sleep. Takes and releases the RCU lock.

Parameters

- **data** – **[in]** pointer to the UMS data
- **args** – **[in]** userspace pointer for args

Returns

- 0 - OK
- -ESRCH - Bad UMS calling thread
- -ENOMEM - No memory available

int **ums_worker_end**(struct *ums_data* *data)

Terminate a UMS worker.

Enqueue a UMS scheduling event of type *THREAD_TERMINATED* to the parent UMS scheduler, destroy the UMS worker and wake up the parent UMS scheduler.

Context: Process context. May sleep. Takes and releases the RCU lock.

Parameters

- **data** – **[in]** pointer to the UMS data

Returns

- 0 - OK
- -ESRCH - Bad UMS calling thread
- -ENOMEM - No memory available

void **ums_worker_destroy**(struct *ums_worker* *worker)

Destroy the UMS worker.

Context: Process context.

Parameters

- **worker** – **[in]** pointer to the UMS worker

4.2.9 Procfs

4.2.9.1 Overview

The module exposes inside the `procfs` several informations about every running scheduler thread and worker thread. In particular for every process that opens the UMS device the following tree is created:

```
/proc/ums/<pid>
├── schedulers
│   ├── ...
│   ├── <scheduler-pid>
│   │   ├── info
│   │   └── workers
│   │       ├── 0 -> /proc/ums/<pid>/workers/<worker-pid>
│   │       ├── ...
│   │       └── n -> /proc/ums/<pid>/workers/<worker-pid>
│   └── ...
└── workers
    ├── ...
    ├── <worker-pid>
    │   └── info
    └── ...
```

Inside the `/proc/ums/<pid>/schedulers` there is a folder for each UMS scheduler thread and inside the `/proc/ums/<pid>/workers` there is a folder for each UMS worker thread.

The `info` file shows some statistics as the pid of the UMS thread, the number of context switches and the current context state (idle/running).

The `/proc/ums/<pid>/schedulers/<scheduler-pid>/workers` folder contains a symbolic link for each UMS worker thread the UMS scheduler has dequeued from the completion list; in particular the links are named from 0 to n and they targets the `/proc/ums/<pid>/workers/<worker-pid>` which they refer to.

4.2.9.2 Structs

struct **ums_proc_dirs**
UMS procfs base directories for each process that opens the UMS device

Public Members

char ***pid_dir_path**
procfs base directory path, i.e. `/proc/ums/<pid>`

size_t **pid_dir_path_size**
procfs base directory path length

struct proc_dir_entry ***pid_dir**
procfs base directory

struct proc_dir_entry ***sched_dir**
procfs schedulers directory located at `/proc/ums/<pid>/schedulers`

char ***workers_dir_path**
procfs workers directory path, i.e. `/proc/ums/<pid>/workers`

size_t **workers_dir_path_size**
procfs workers directory path length

struct proc_dir_entry ***workers_dir**
procfs workers directory

struct **ums_scheduler_proc_dirs**
UMS scheduler procfs directories

Public Members

struct proc_dir_entry ***scheduler_dir**
procfs scheduler directory located at `/proc/ums/<pid>/schedulers/<sched-pid>`

struct proc_dir_entry ***scheduler_info_dir**
scheduler info file located at `/proc/ums/<pid>/schedulers/<sched-pid>/info`

struct proc_dir_entry ***workers_dir**
procfs scheduler's workers directory located at `/proc/ums/<pid>/schedulers/<sched-pid>/workers`

unsigned long **n_workers**
number of UMS workers dequeued from the UMS scheduler's completion list

unsigned long **max_workers**

struct **ums_worker_complist_node**
UMS completion list node with UMS worker

Public Members

struct proc_dir_entry ***worker_link**
Symbolic link targeting the UMS worker

struct **ums_worker_proc_dirs**
UMS worker procfs directories

Public Members

char *`worker_dir_path`
: procfs worker directory path, i.e. `/proc/ums/<pid>/workers/<worker-pid>`

size_t `worker_dir_path_size`
: procfs worker directory path length

struct proc_dir_entry *`worker_dir`
: procfs worker directory

struct proc_dir_entry *`worker_info_dir`
: worker info file located at `/proc/ums/<pid>/workers/<worker-pid>/info`

struct *ums_worker_complist_node* `complist_node`
: procfs worker node with a completion list

4.2.9.3 Functions

int `ums_proc_init`(void)
: Initialize the UMS procfs.

  Returns

  - 0 - OK
  - -ENOMEM - No memory available

void `ums_proc_destroy`(void)
: Destroy the UMS procfs.

int `ums_proc_dirs_init`(struct *ums_proc_dirs* *dirs)
: Initialize the UMS procfs base directories.

  Parameters

  - **dirs** – **[in]** pointer to the UMS procfs directories

  Returns

  - 0 - OK
  - -ENOMEM - No memory available

void `ums_proc_dirs_destroy`(struct *ums_proc_dirs* *dirs)
: Destroy the UMS procfs base directories.

  Parameters

  - **dirs** – **[in]** pointer to the UMS procfs directories

const char *`get_context_state`(struct *ums_context* *context)
: Get a string representation of the UMS context state.

  Parameters

  - **context** – **[in]** pointer to the UMS context

Returns the context state representation

static inline int **context_snprintf**(char *buf, size_t size, struct *ums_context* *context)

Get a string representation of the UMS context.

Parameters

- **buf** – **[inout]** pointer to an allocated buffer
- **size** – **[in]** maximum number of bytes to write
- **context** – **[in]** pointer to the UMS context

Returns the number of characters printed or truncated

int **ums_scheduler_proc_register**(struct *ums_proc_dirs* *dirs, struct *ums_scheduler* *scheduler)

Register the UMS scheduler to the UMS procfs.

Parameters

- **dirs** – **[inout]** pointer to the UMS procfs directories
- **scheduler** – **[in]** pointer to the UMS scheduler

Returns

- 0 - OK
- -ENOMEM - No memory available

void **ums_scheduler_proc_unregister**(struct *ums_scheduler* *scheduler)

Unregister the UMS scheduler to the UMS procfs.

Parameters

- **scheduler** – **[in]** pointer to the UMS scheduler

int **ums_scheduler_proc_register_worker**(struct *ums_scheduler_proc_dirs* *dirs, struct *ums_worker_proc_dirs* *worker_dirs)

Register the UMS worker to the UMS scheduler procfs.

Create a symlink inside `/proc/ums/<pid>/schedulers/<sched-pid>/workers` with the UMS worker as target.

Parameters

- **dirs** – **[inout]** pointer to the UMS scheduler procfs directories
- **worker_dirs** – **[inout]** pointer to the UMS worker procfs directories

Returns

- 0 - OK
- -ENOMEM - No memory available

void **ums_scheduler_proc_unregister_worker**(struct *ums_scheduler_proc_dirs* *dirs, struct *ums_worker_proc_dirs* *worker_dirs)

Unregister the UMS worker to the UMS scheduler procfs.

Remove the symlink inside `/proc/ums/<pid>/schedulers/<sched-pid>/workers` with the UMS worker as target.

Parameters

- **`dirs`** – **[inout]** pointer to the UMS scheduler procfs directories
- **`worker_dirs`** – **[inout]** pointer to the UMS worker procfs directories

int **`ums_worker_proc_register`**(struct *ums_proc_dirs* *dirs, struct *ums_worker* *worker)

Register the UMS worker to the UMS procfs.

Parameters

- **`dirs`** – **[inout]** pointer to the UMS procfs directories
- **`worker`** – **[in]** pointer to the UMS worker

Returns

- 0 - OK
- -ENOMEM - No memory available

void **`ums_worker_proc_unregister`**(struct *ums_worker* *worker)

Unregister the UMS worker to the UMS procfs.

Parameters

- **`worker`** – **[in]** pointer to the UMS worker

CHAPTER

FIVE

TOOLS

5.1 Benchmark tool

If the package has been build with `--enable-benchmark` option then the following tool is provided for executing a benchmark of the UMS mode with respect to the default pthread one:

```
$ ums-benchmark [OPTION]... <ums|pthread>
```

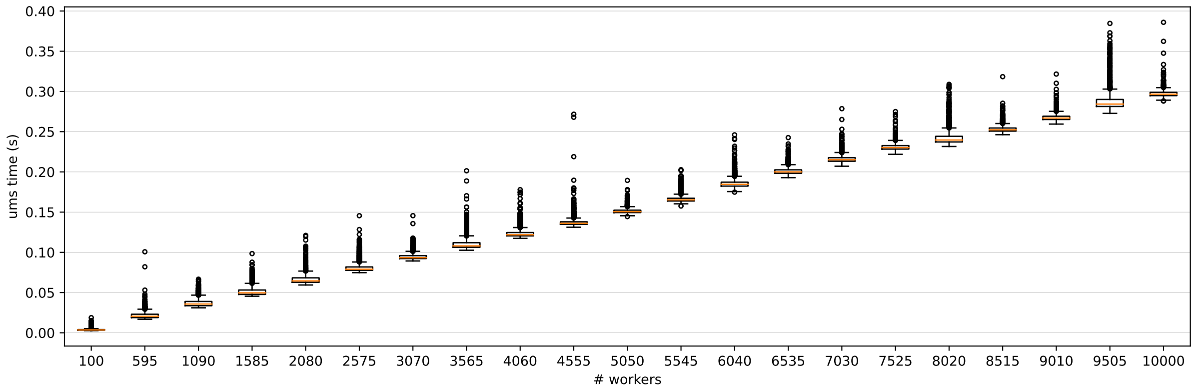
5.1.1 Options

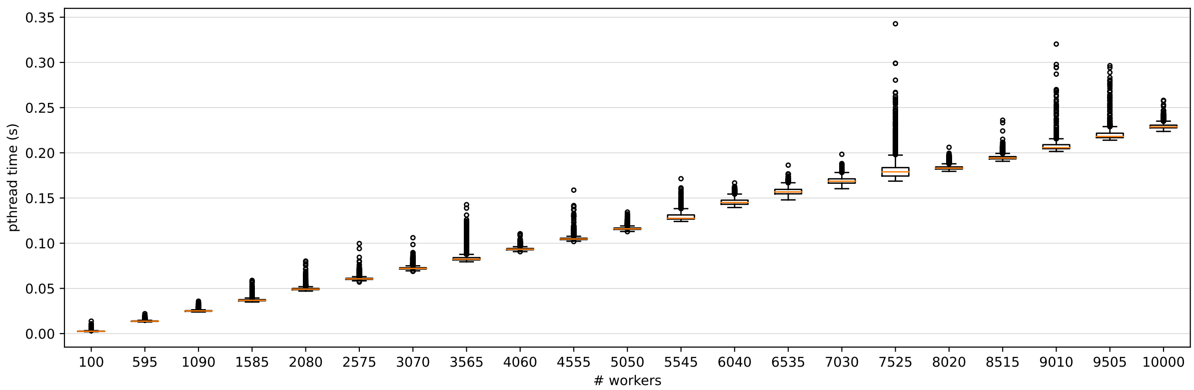
`-t, --task=TASK`	Benchmark task to execute; default to `prime`
`-w, --workers=NUM`	Number of UMS worker threads to run

BENCHMARKS

The following results are obtained on a Hyper-Threading *Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz* with 6 cores (12 threads).

The benchmark program[1] consists of a set of UMS schedulers (one per-CPU core) managing a set of UMS worker threads and all sharing a single completion list. Each UMS worker thread compute the same task of executing a primality test over a 16-bit prime number.





The UMS library solution performs slightly worse than the system default one due to the fact that a mode switch is needed for executing and yielding a UMS context.

[1] source code at `benchmark/bin/src/main.c`

INDEX

Symbols

A

C

D

E

F

G

I

P

R

S

U

W