# Advanced Operating Systems and Virtualization

Fiber project

**Matteo Mariani** 1815188

**Riccardo Chiaretti** 1661390

# Contents

**01. Introduction** **3**

**02. Module** **3**

Core data structures . . . . . 3

`ioctl.c` . . . . . 4

`fiber.c` . . . . . 5

`ConvertThreadToFiber()` . . . . . 5

`CreateFiber()` . . . . . 5

`SwitchToFiber()` . . . . . 6

`FlsAlloc()` . . . . . 6

`FlsSet()` . . . . . 6

`FlsGet()` . . . . . 6

`FlsFree()` . . . . . 6

`kprobe.c` . . . . . 7

`proc` filesystem . . . . . 8

**03. Comparisons** **9**

**A. Appendix** **11**

# 01. Introduction

The following report discusses about the implementation of Kernel-level fibers into Linux OS. The concept of *fiber* was born in Windows OS where they are already implemented as lightweight User-level threads. More in depth, a fiber is a unit of execution that runs in the context of a thread and it must be explicitly scheduled by the thread itself.[1] A thread can take advantage of this mechanism only after having called a specific function. From now on, the calling thread is now the currently executing fiber. In general, a thread can have multiple fibers that share the same `tgid`.

Our goal is implementing the same mechanism but in Kernel-level.

We developed two variants of the project that differ in the way according to which fiber subsystem is represented. In the following analysis we'll use the version that reached the best performance, using the other for a metric of comparison in the dedicated section.

# 02. Module

In order to accomplish our goal and provide the required functionalities, we can follow two strategies:

- implement a set of system calls that encapsulate the needed services and recompile the kernel to have a default and dedicated subsystem;
- write a loadable kernel module (LKM) that wraps up the necessary operations without the need to recompile the kernel. This strategy needs a way to interface the module with the User-space - using `ioctl()` system call.

For our project, we decided to use the second approach.

Once the kernel module is loaded, a device is created in `/dev` which is then opened every time that fiber subsystem needs to be used. Moreover, some kprobes are registered to handle particular situations that will be described in the following analysis.

## Core data structures

The main data structures are: `process_t`, `thread_t` and `fiber_t`.

```
typedef struct {
    pid_t tgid;
    atomic_t total_fibers;
    atomic_t total_threads;
    DECLARE_HASHTABLE(Fibers, 10);
    DECLARE_HASHTABLE(Threads, 10);
    struct hlist_node table_node;
} process_t;
```

`process_t` data structure represents the set of threads, having the same `tgid`, which issued `ConvertThreadToFiber()` function. It also stores information about the number of total threads that issued the previous function (`total_threads`) and the number of total fibers (`total_fibers`) managed by those threads.
Each `process_t` is stored in a global hashtable called `Processes`.

[1] A fiber is non-preemptive and it never exits. Indeed, if a fiber terminates then the whole thread exits as well.

```
typedef struct {
    pid_t pid;
    fiber_t* running_fiber;
    struct hlist_node table_node;
} thread_t;
```

`thread_t` data structure represents a single thread that is now a fiber. It is identified by its `pid` and it has a pointer to the fiber which the thread is currently running (`*running_fiber`).

```
typedef struct {
    spinlock_t lock;
    char running;
    int fiber_id;
    int activations;
    int failed_activations;
    pid_t parent_pid;
    unsigned long exec_time;
    unsigned long start_time;
    long long fls[MAX_FLS];
    struct pt_regs *context;
    struct fpu *fpu_regs;
    void *initial_entry_point;
    char name[512];
    DECLARE_BITMAP(fls_bitmap, MAX_FLS);
    struct hlist_node table_node;
} fiber_t;
```

`fiber_t` data structure represents a single fiber that can be run by a single thread. It is identified by its `fiber_id` and `parent_pid` - that is the creator thread.
Considering a multi-threaded scenario, the access to this data structure is safe thanks of the usage of the `spinlock_t lock`: if someone wants to switch to a fiber, it tries to acquire the lock for finalizing the operation.
Since the project requires to evaluate the performances, some useful information were introduced:

- `activations` keeps the number of finalized switches to the fiber;
- `failed_activations` keeps the number of failed tentatives to switch to the fiber;
- `start_time` stores the time in which the fiber started running in a single session of activation, expressed in nanoseconds;
- `exec_time` stores the total execution time of the fiber during its whole lifetime.

In order to store the context of each fiber once yielded the CPU, `context` is used to save a snapshot of the CPU registers while `fpu_regs` holds the content of fpu registers.
Since a single fiber shares the address space with other fibers as well as other kernel level threads sharing the same `tgid`, we need a way to let a single fiber to have its own local memory space. An FLS (Fiber Local Storage) is a reserved memory area used by the fiber to store private data. To manage this, there are a buffer `fls[MAX_FLS]` and an associated bitmap `fls_bitmap` to keep track of its content.
Lastly, `initial_entry_point` represents the memory address of the function associated to the fiber.

## ioctl.c

The `ioctl.c` file exports all the functionalities implemented by the module. Once the user space issues a call to the fiber subsystem, this is done through an `ioctl` call specifying the appropriate ioctl command numbers. There are seven different command numbers, each of them associated to a specific exposed functionality.

The revisited implementation of the original `ioctl` system call is `my_ioctl`.
To make possible for the user space to pass parameters to kernel space - and viceversa - it has been defined an `ioctl_params` to contain the required data. The copy is done through `copy_from_user` and `copy_to_user`.

## fiber.c

The `fiber.c` file contains the implementation of all core functions: those are called directly from `ioctl.c`.

### ConvertThreadToFiber()

`ConvertThreadToFiber()` is in charge of creating all the data structures needed for the calling thread to take advantage of the fiber subsystem. This function needs to be called each time a thread wants to convert to a fiber.
In particular, it does the following:

1. it checks whether already exists some thread sharing the same `tgid` of the caller. If this is not the case, a new `process_t` data structure is created and added to the `Processes` hashtable by a call to `init_process`;
2. then, it creates a new `thread_t` data structure for the calling thread as well as a new `fiber_t` data structure. Note that the new fiber inherits the same CPU context of the calling thread. Its `initial_entry_point` is set to the IP of the thread at its calling time;
3. once the fiber is created, it is considered running by the calling thread.

In order to avoid concurrent creation of the same `process_t` data structure, the `spinlock_t process_lock` is used to safely check whether there exists or not an entry belonging to the `tgid` of the caller. In order to reduce the critical section, the lock is released soon after the check and the creation of the entry in the hashtable.
The `convertThreadToFiber()` returns the `fiber_id` of the new fiber.

### CreateFiber()

`CreateFiber()` is responsible for creating a new `fiber_t` data structure which is inserted in the pool of fibers associated to the `tgid` of the caller.
This function can be called only if the caller already issued `ConvertThreadToFiber()`.
In particular, it does the following:

- it sets the IP of the fiber's context (`new_fiber->context->ip`) to the `initial_entry_point` passed by user space. This corresponds to a function from which the fiber starts executing;
- it sets BP and SP to the memory area passed by user space corresponding to the stack of the new fiber;
- it sets DI to the data structure containing the formal parameters of the function corresponding to the initial entry point.

The `CreateFiber()` returns the `fiber_id` of the new fiber.

#### SwitchToFiber()

`SwitchToFiber()` is in charge of managing the context switch between the currently running fiber and the target fiber provided by user space specifying its id.
This function can be called only by a thread that is already a fiber and if the target fiber exists but is not currently running. Note that the caller thread can switch to all the fibers created by another thread sharing the same `tgid`.
In order to avoid that multiple fibers try to concurrently schedule the same target fiber creating possible inconsistency in the system, the `target_fiber->lock` is acquired by the caller before attempting the switch.
What the function does is the following:

1. the snapshot of the CPU is copied into the fiber `current_fiber` which is going to be de-scheduled;
2. the `exec_time` of the `current_fiber` is updated;
3. the context of the target fiber `target_fiber` is "copied" into the CPU;[2]
4. the `start_time` of the `target_fiber` is set;
5. the `running_fiber` pointer of the caller is updated pointing to the new currently running fiber.

Once this procedure is completed, `target_fiber->lock` is released.

#### FlsAlloc()

`FlsAlloc()` function is in charge of finding one memory location in the `fls` where the running fiber can store data.
In order to enhance the performances, it is used a bitmap (`fls_bitmap`) to rapidly check the first available position in the buffer. For this reason, `FlsAlloc()` uses the bitmap to return the index `pos` of the `fls` in which data can be stored.

#### FlsSet()

`FlsSet()` function stores the `value` passed from user space in the `pos` position returned by `FlsAlloc()`.

#### FlsGet()

`FlsGet()` function returns the value stored in the buffer `fls` at position `pos` passed by user space. The value can be retrieved only if the bit in position `pos` in `fls_bitmap` is set 1, otherwise it is considered as an illegal action.

#### FlsFree()

`FlsFree()` function frees the position `pos` in the `fls` of the current fiber. In order to do so, the bit in position `pos` in `fls_bitmap` is set to 0.

[2] It is actually copied in the memory area returned by `task_pt_regs`. Then it is up to the Kernel to put the information in that memory area into the registers.

### kprobe.c

In `kprobe.c` there are multiple functions that register Kprobe and Kretprobe. Those are used to perform a particular task depending on the hooked function. Both of them hook a provided function, intercepting the call in precise points and executing the code defined in a Pre Handler and Post Handler. The first handler is executed *before* the execution of the probed function. The execution of the second handler differ depending on the type of the Probe:

- Kprobe executes the Post Handler *before* the `ret` instruction;
- Kretprobe executes the Post Handler *after* the `ret` instruction.

In our case, there are two situations that need to be handled properly:

- when a thread exits there is the need to update the fiber currently run by that thread. Indeed if this is not done, that fiber will never be scheduled again. Moreover, data structures stored in the hashtables need to be freed;
- when a thread is de-scheduled by the scheduler, the execution time of its running fiber should be updated. This is fundamental for statistics.

In order to solve these problems, we decided to use Kprobes and Kretprobes on `do_exit()` and `finish_task_switch()` functions respectively.

- `do_exit()`. If the exiting thread belongs to the fiber subsystem then the `total_threads` number associated to its `tgid` is decremented. Moreover, its `running_fiber` becomes available for future switches and the `exec_time` is updated. However, if the exiting thread is the last one with that particular `tgid`, all data structure related to the `process_t` are deleted from the `Processes` hashtable;
- `finish_task_switch()`. We defined a per CPU variable `prev` in order to keep track of the thread that is currently on each CPU. This was done to understand which thread was running in the CPU at the moment when `finish_task_switch()` was called. Once this information is collected, the `exec_time` and `start_time` of both fibers (the one of the de-scheduled thread and the one of the scheduled thread) is updated.

## proc filesystem

An additional requirement of the project is to create a folder `fiber` for each thread participating in the fiber subsystem into `proc` filesystem. In every `/proc/<pid>/fiber` folder there is one file `<fiber_id>` for each fiber created by the threads sharing the same `tgid`.
In each `<fiber_id>` file there are useful runtime information about the fiber.

```
matteo@ubuntu:/proc/6526/fiber$ cat 95
Running: no
Initial entry point: 0x0000560cff3d06a0
Parent thread id: 6526
Number of activations: 1
Number of failed activations: 0
Total execution time: 546 ms
```

All the elements in `/proc/<pid>` are contained into a static and constant array named `tgid_base_stuff`. Since we want to modify the number of entries of that array, we have to find a way to do so.
Our solution involved the use of Kprobes on some functions called when `ls` and `cd` commands are called inside `/proc`. Those functions are `proc_tgid_base_readdir` and `proc_tgid_base_lookup`, respectively. Those functions wrap other calls to `proc_pident_readdir` and `proc_pident_lookup`, respectively.
Let's consider the case of `proc_tgid_base_readdir` - since the other one was treated in a similar way. Our goal was to insert the `<fiber>` folder, so we decided to use a Kretprobe in such a way to let the probed function work as intended and then exploit the Post Handler in order to insert the new folder. Before its ending, the Post Handler will call the function `proc_pident_readdir` with the new parameters necessary to masquerade the results.

Some inode operations and file operations related to `/proc/<pid>/fiber` folders have been changed in order to support proper navigation throughout these folders. We implemented: `fiberReaddir` to handle `ls` command and `fiberLookup` to handle `cd` command.

```
struct file_operations file_ops = {
        .read  = generic_read_dir,
        .iterate_shared = fiberReaddir,
        .llseek  = generic_file_llseek,
};

struct inode_operations inode_ops = {
        .lookup = fiberLookup,
};
```

`fiberReaddir` and `fiberLookup` differ from the default functions because they scan our data structure in order to populate the `<fiber>` folder with the `<fiber_id>` files.

Secondly, we implemented a function called `fiberRead` to show the runtime information associated to each `<fiber_id>` file. This is necessary when `cat` commands is done on these files.

```
struct file_operations fiber_ops = {
        read: fiberRead,
};
```

Indeed, what `fiberRead` does is to retrieve the information stored from the `fiber_t` data structure and display them, populating the `<fiber_id>` file with these information at runtime.

# 03. Comparisons

As introduced before, we developed two variants of the fiber project:

- the first version is the one described throughout the entire analysis. We have an hashtable `Processes` that contains the `process_t` data structure, one for each process. A `process_t` data structure contains two hashtable: `Threads` and `Fibers`. The first one consists of all `thread_t` data structure representing all the threads that share the same `tgid`. The second one represents the pool of fibers `fiber_t` created by those threads. Moreover, each thread has its own `*running_fiber` that points to the fiber that it is currently running;
- the second version drops the `thread_t` data structure and the correspondent hashtable `Threads` to use only `Processes` and `Fibers` ones. Basically, one `process_t` directly manages the fibers that are created by its threads.

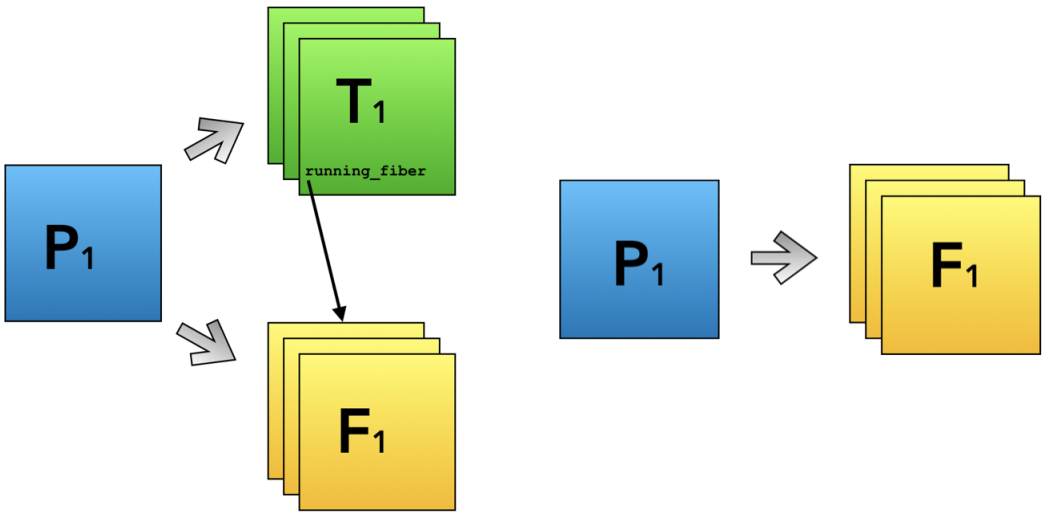
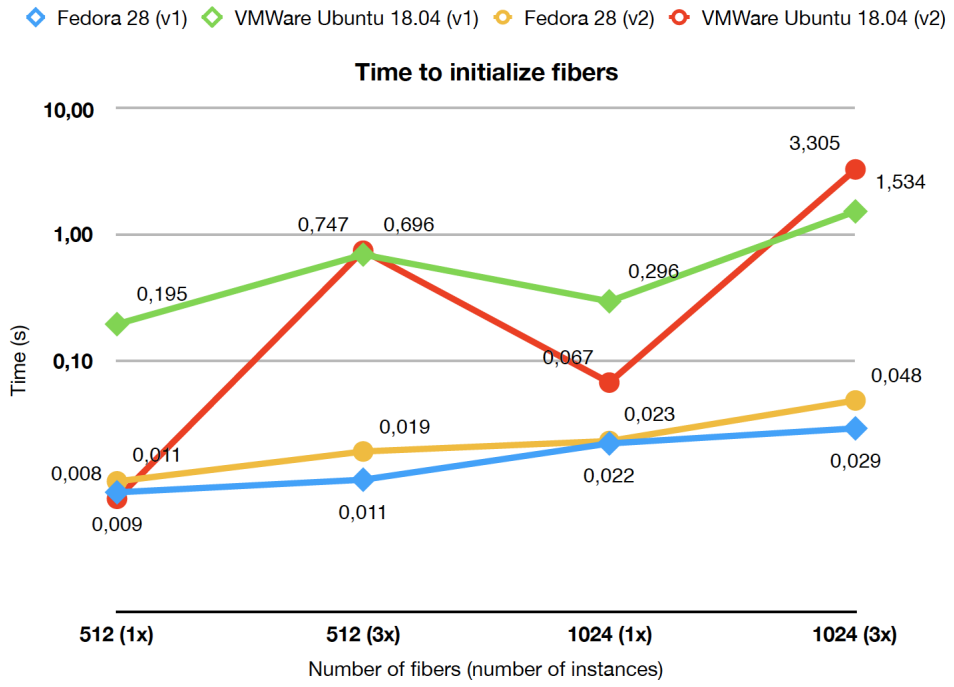
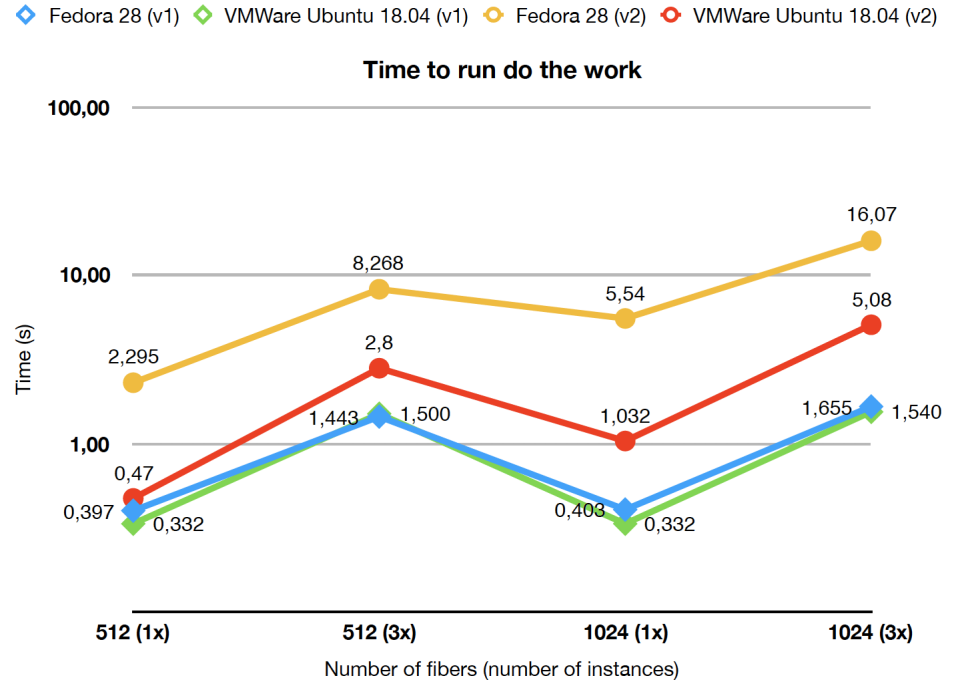



Figure 1: *Version 1* - left figure. For each `tgid` in the subsystem there are two hashtables: one for the threads and one for the fibers belonging to the same `tgid`. *Version 2* - right figure. For each `tgid` in the subsystem there is a hashtable containing all the fibers associated to that `tgid`.

The benchmark was done using (i) Fedora 28 (physical machine) and (ii) Ubuntu 18.04 run by VMWare (virtual machine) - see *A. Appendix* for further details. Firstly we tested how both versions handle 512 and 1024 fibers then we tested the concurrency aspect by launching the benchmark on multiple terminals at the same time - three terminals with 512 and 1024 fibers.

The results were the following:




The main drawback of Version 2 is related to the low performances achieved in the most used function in the system, that is `SwitchToFiber`. Indeed, since we do not have the possibility to directly retrieve the currently running fiber of a thread (`thread_t->running_fiber`) - due to the lack of the concept of a thread in the fiber system - we had to scan the whole `Fibers` hashtable to search for it.[3] In the worst case, the cost of the search operation would have been $O(n)$, where $n$ is the number of fibers for each `tgid`.

[3] We had to use `hash_for_each_rcu` instead of `hash_for_each_possible_rcu`.

# A. Appendix

The following two figures represent a summary of the specs of the machines used for testing.
Fedora 28 was run as physical machine, Ubuntu 18.04 was run as virtual machine with VMWare.