# Fibers implementation report

Advanced Operating Systems and Virtualization, A.A 2017/2018

Maria Ludovica Costagliola
*costagliola.1657716@studenti.uniroma1.it*

Emanuele De Santis
*desantis.1664777@studenti.uniroma1.it*

## 1 Introduction

This essay discusses the implementation of Fibers, made as final project for the *Advanced Operating Systems and Virtualization* course. Fibers correspond to User-Level Threads and in Windows OS they are implemented at kernel-level. Through this project, we took care of inserting this functionality inside the Linux kernel. To accomplish it, we have developed a kernel module that implements all functionalities needed to support their execution.

A Fiber can be seen as a lightweight thread of execution. Unlike usual kernel-level threads, Fibers have to explicitly yield the execution to let another fiber run.

### *MODULE*

When the module is loaded, it will register the device that is needed to let the paradigm of IOCTL work, through `register_fiber_device()`, and several kprobes to have a sound and complete implementation of Fibers.

At the unloading of the module, there is a cleanup function that takes care of unregistering all kprobes and removing the device file used for IOCTL.

## 2 Kernel Level

We developed the whole logic of Fibers at kernel level and made accessible to user-level the minimal amount of information possible.

### *DATA STRUCTURES*

We used three main data structures to handle Fibers and related information:

- `struct process`
- `struct thread`
- `struct fiber`

The `struct process` is unique inside the process and is initialized by the first thread of the process that converts to a fiber: we register here the number of active threads of the process, the number of fibers created until that point, two hashtables to keep trace of all converted threads and of all fibers created by whatever thread of the process. There are also the `process_id` and a `struct hlist_node`, both needed to insert this process inside a global hashtable that contains all active processes that are using Fibers.

```
struct process {
        pid_t process_id; //key for the hashtable processes
        struct hlist_node node;
        atomic_long_t last_fiber_id;
        atomic_long_t active_threads;
        DECLARE_HASHTABLE(threads, 10);
        DECLARE_HASHTABLE(fibers, 10);
};
```

Each thread that wants to convert to a fiber will initialize a `struct thread` that contains `thread_id` and a `struct hlist_node` to be inserted in the hashtable of the parent process; there is a pointer to the `struct process` of its parent process and a pointer to the fiber that it is currently running.

```
struct thread {
        pid_t thread_id; //key for the hashtable threads
        struct hlist_node node;
        struct process *parent;
        struct fiber *selected_fiber;
};
```

Finally, the fundamental data structure is the `struct fiber`, one for each fiber. In this struct there is a lock used to serialize threads requests to switch to that fiber. There is also a pointer to the parent process and the thread executing this fiber. The struct contains a `struct pt_regs` and `struct fpu` to be able to save and restore its execution context. In the struct there are also starting address of the fiber's stack and its size. To handle FLS (*Fiber Local Storage*), there is an array of fixed size `MAX_FLS_POINTERS`, and a bitmap to quickly find available cells inside the array. The last fields in this data structure are needed to compute some statistics to be shown inside proc subsystem.

```
struct fiber {
        //here we have to put all the fields that we want to use in a
           fiber
        //for example pointer to the stack, saved registers, locks
        //and all the other information we need.
        spinlock_t fiber_lock;
        unsigned long flags;
        pid_t fiber_id; //key for the hashtable fibers
        char name[256];
        struct hlist_node node;
        struct thread *attached_thread; //NULL if no thread executes
           this fiber
        struct process *parent_process;


        //CPU context
        struct pt_regs registers; //copy of the pt_regs struct that
           points into the kernel level stack
        struct fpu fpu; // to replace in task_struct->struct_thread->
           fpu upon context switch

        void *fiber_stack;
        unsigned long fiber_stack_size;
```

```
        long long fls[MAX_FLS_POINTERS];
        DECLARE_BITMAP(fls_bitmap, MAX_FLS_POINTERS);

        //some statistics...
        void* start_address;
        pid_t creator_thread;
        unsigned long activation_counter;
        atomic_long_t failed_activation_counter;
        unsigned long total_time;

};
```

## *IOCTL*

The file `ioctl.c` comes into play each time is issued an IOCTL call at user level. We have seven IOCTL commands, one for each functionality the module has to handle, that are defined inside `ioctl.h` and are associated to a different number, from 0 to 6.

The registered function is `fibers_ioctl()`. It checks the IOCTL command issued and calls the appropriate function to perform the desidered task. If the user was supposed to pass some parameters (wrapped in a `struct fiber_arguments`), there is a check on this structure (to verify that the address passed is a valid one) and then it is copied inside a local structure. At the end, we call the actual function that performs the task requested by the user.

In case of a switch, we pay attention to take a lock right before calling the actual switch function and release it as soon as this function returns. This mechanism is used since we are in an SMP context to be safe and secured.

If the user issues an IOCTL call to get a value inside the fiber local storage, then it is important to make a `copy_to_user` with this just retrieved value, since the user-level stub for IOCTL returns an int (to maintain compatibility with the old signature of IOCTL handler).

The actual functions are implemented inside `fibers.c`.

## *CONVERT A THREAD TO A FIBER*

In order to use Fibers' logic, it's important to have a first fiber to start with. The function `do_ConvertThreadToFiber` is called by each thread that wants to use Fibers for the first time. The very first thread in a process that calls it, has to initialize the `struct process` data structure related to the process the thread lives in.

`do_ConvertThreadToFiber` has to create the `struct process` only if it doesn't yet exists. To do so in a SMP context it uses a lock.

After that, it creates also a `struct thread` representing the thread and a `struct fiber` representing the first fiber for that thread. This *special* fiber uses the current value of the `rip` register as starting point and is automatically marked as running by that thread.

This function returns the id of the newly created fiber.

## *CREATE A NEW FIBER*

Using this funtionality, a thread has the possibility to create a new fiber, assigning to that a starting address (usually a function pointer) and a parameter to be passed.

In our module this functionality is implemented through `do_CreateFiber`. This function looks for the

process and the thread in the respective hashtables (if it doesn't find even one of them, it means that the calling thread has not perfomed the `ConvertThreadToFiber` call yet).

After these checks, it allocates a new `struct fiber`, computes the stack pointer using the base stack address and the stack size and adds the struct to the process's hashtable.

It then returns the id of the newly created fiber.

## *SWITCH TO A FIBER*

Since Fibers are not preemptable, they have to yield the execution to let another fiber run. To do so our module uses the `do_SwitchToFiber` function.

This function first checks if the selected fiber is not running (in fact if it is already running it has to fail) and cares to save the CPU and FPU status in the `struct fiber` related to the previous fiber and to restore the CPU and FPU context taken form the next `struct fiber`.

It has also to reflect the switch in the control pointers both of the `struct thread` and of the `struct fiber`.

## *FIBER LOCAL STORAGE*

Each fiber shares the address space with each other fiber and each other normal kernel thread. To let a fiber have its own local space, we implement a Fiber Local Storage as an array of `long long` of fixed size. There are 4 APIs to interact with this local storage:

- `do_FlsAlloc`
- `do_FlsSetValue`
- `do_FlsGetValue`
- `do_FlsFree`

In this way, each fiber can allocate a cell, set a value for that cell, read that value and free the cell. To better perform these operations we have used a bitmap: in this way we can find a free space in $O(\log n)$ instead of $O(n)$.

Each function checks the validity of the arguments passed to be always safe.

## *KPROBES*

Our module needs a kprobe to release all the memory used once a thread or the whole process die. This kprobe is executed each time `do_exit()` is called. Since `do_exit()` is called when the current thread is about to die, this is the right place to free all our structures regarding that thread. Moreover, if the current thread is the last one in the process still executing fibers, this leads to a complete cleanup of all `struct fiber` belonging to that process and the `struct process` itself.

We also rely on kprobes to patch the `/proc/{PID}` directory to show up also the `fibers` folder only in those processes that have at least one fiber.

We use a third kprobe to update fiber's execution time whenever the system schedule runs.

### *PROC SUBSYSTEM*

Each fiber stores some information needed to compute runtime statistics about its usage. These statistics are displayed in a pseudofile named with the fiber id in `/proc/{PID}/fibers`.

`/proc/{PID}` has a very particular structure, different with respect to a normal proc directory/file. In fact, all elements appearing in `/proc/{PID}` are defined in a static constant array (in `/fs/proc/base.c`) named `tgid_base_stuff`. To add the folder `fibers` to `/proc/{PID}`, we had to modify the behaviour of the `iterate_shared` handler (in this case `proc_tgid_base_readdir`) and the `lookup` handler (in this case `proc_tgid_base_lookup`). We did it using two kretprobes.

We examine the case of the `iterate_shared` since the other one uses almost the same approach. In this case we compute once (as soon as the kretprobe is fired for the first time) the number of entries added by the actual `proc_pident_readdir`, called by `proc_tgid_base_readdir`, and then we call again `proc_pident_readdir` to insert an ad-hoc array containing only the `fibers` entry. In this way the inode for our entry is instantiated exactly like the ones corresponding to the actual entries of `tgid_base_stuff` and the `dir_context` is update accordingly.

To show under `/proc/{PID}/fibers` all the pseudofiles corresponding to the fiber instantiated in that process, we used the same approach of `tgid_base_stuff` and `proc_pident_readdir`, but this time the array is not a static constant array, but it is dynamically generated based on the fibers present in the process.

Each of these files has the same `read` handler, that extract information from its argument `struct file`, to understand which fiber's information outputs to user-level.

## 3 User Level

To make visible to user-level applications all interfaces we did in the kernel code, we developed an user-level library that is able to take the user-level application's call and redirect in the correct way to the kernel module using IOCTL.

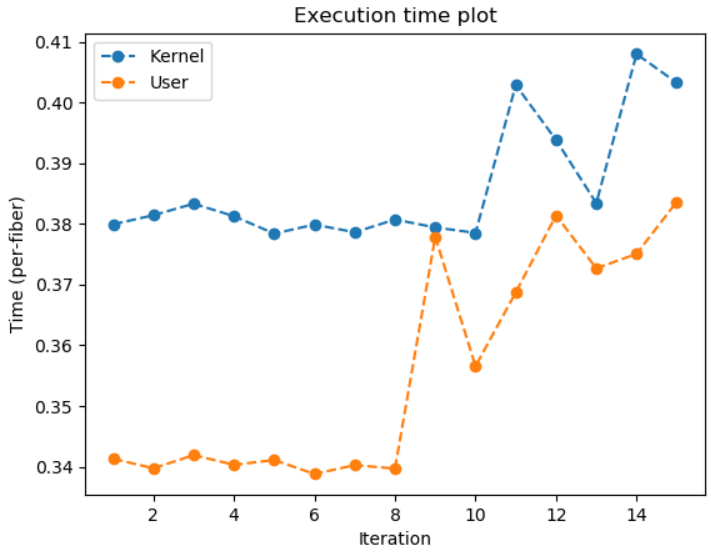
Since we use a dynamically assigned major number for the device `/dev/fibers` to compute IOCTL commands' numbers, we had to find a way to pass these numbers to user-space. The solution was to implement a `read` handler for `/dev/fibers` (besides the `unlocked_ioctl`) that outputs the IOCTL commands' numbers in a predefined scheme.

The user-space library registers a constructor that opens a file descriptor to `/dev/fibers`, reads on it and stores the numbers, following the predefined scheme.

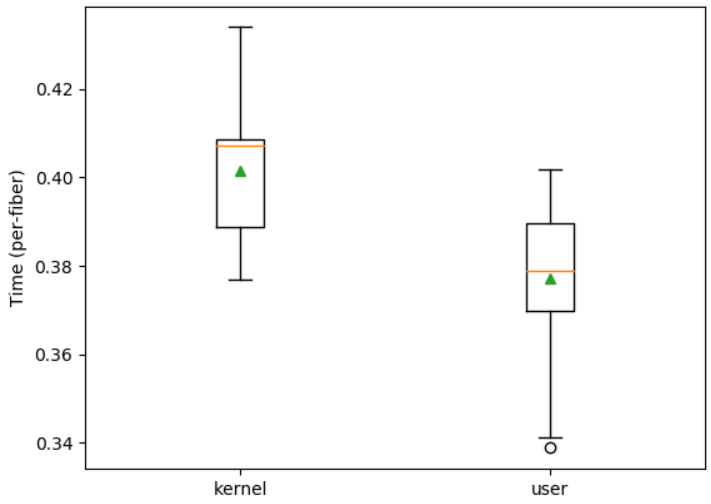
Moreover it wraps all the IOCTL calls to hide the complexity to the user and to pass the data the user provides in the correct manner.

## 4 Performances

Using the given benchmark application, we was able to perform a basic performance test of our module with respect to the user-space implementation. In the first case we run 15 iteration with a growing number of fibers (each time 32 fibers more than the previous time). In the second case we run again 15 iteration of the test application with 550 fibers running and we collected the results.
We used PyPlot to plot all the data collected.




As we can see in this first plot, it is clear that our module usually takes more time with respect to the user-space implementation. This is because of the context switches from user-space to kernel-space and vice versa. We can also notice a deterioration of the performances as the time goes on caused by CPU throttling (in fact the benchmark generates a lot of CPU work, so the CPU temperature increases).
Anyway, it is possible to see that kernel level implementation runs approximately in constant time with respect to the number of fibers (in fact the kernel module is called approximately $O(n)$ times and the per-fiber time remains almost the same).




These boxplots show us a more compact view over all iterations both for kernel-space implementation and for user-space implementation fixing the number of fibers to 550. It shows for each implementation the quantiles (black horizontal lines), the median (yellow horizontal line) and the mean (green triangle) given from the samples we took during the iterations.
From this view it possible to clearly see that, except for outliers, the majority of the kernel-implementation times are very close to the user-implementation times, so the median kernel overhead is very small.