# Final Project Report

Advanced Operating System and Virtualization, A.Y. 2017/2018

Dario Litardi

litardi.1489178@studenti.uniroma1.it

Marco Cuoci

cuoci.1630470@studenti.uniroma1.it

# Summary

# 1. Introduction

This essay discusses about the design choices and the implementation details of the final project. Fibers are the Windows kernel-level implementation of User-Level Threads. A fiber is a unit of execution that must be manually scheduled by the application. Fibers run in the context of the threads that schedule them. Each thread can schedule multiple fibers.

Fibers are not preemptively scheduled. You schedule a fiber by switching to it from another fiber. The system still schedules threads to run. When a thread running fibers is preempted, its currently running fiber is preempted but remains selected. The selected fiber runs when its thread runs.

We have choose to implement the fibers like a kernel module. The module is a character device file.

The module is loaded through the function *fib_driver_init()* and unmounted with *fib_driver_exit()*. The function *fib_driver_init()* creates the device and registers it; moreover it registers the kprobes.

The function *fib_driver_exit()* unregisters the device and the kprobe structures.

We have developed on the kernel linux's version 4.9.0-8-amd64 with the operating system Debian GNU/Linux 9.

# 2. Kernel Level Design

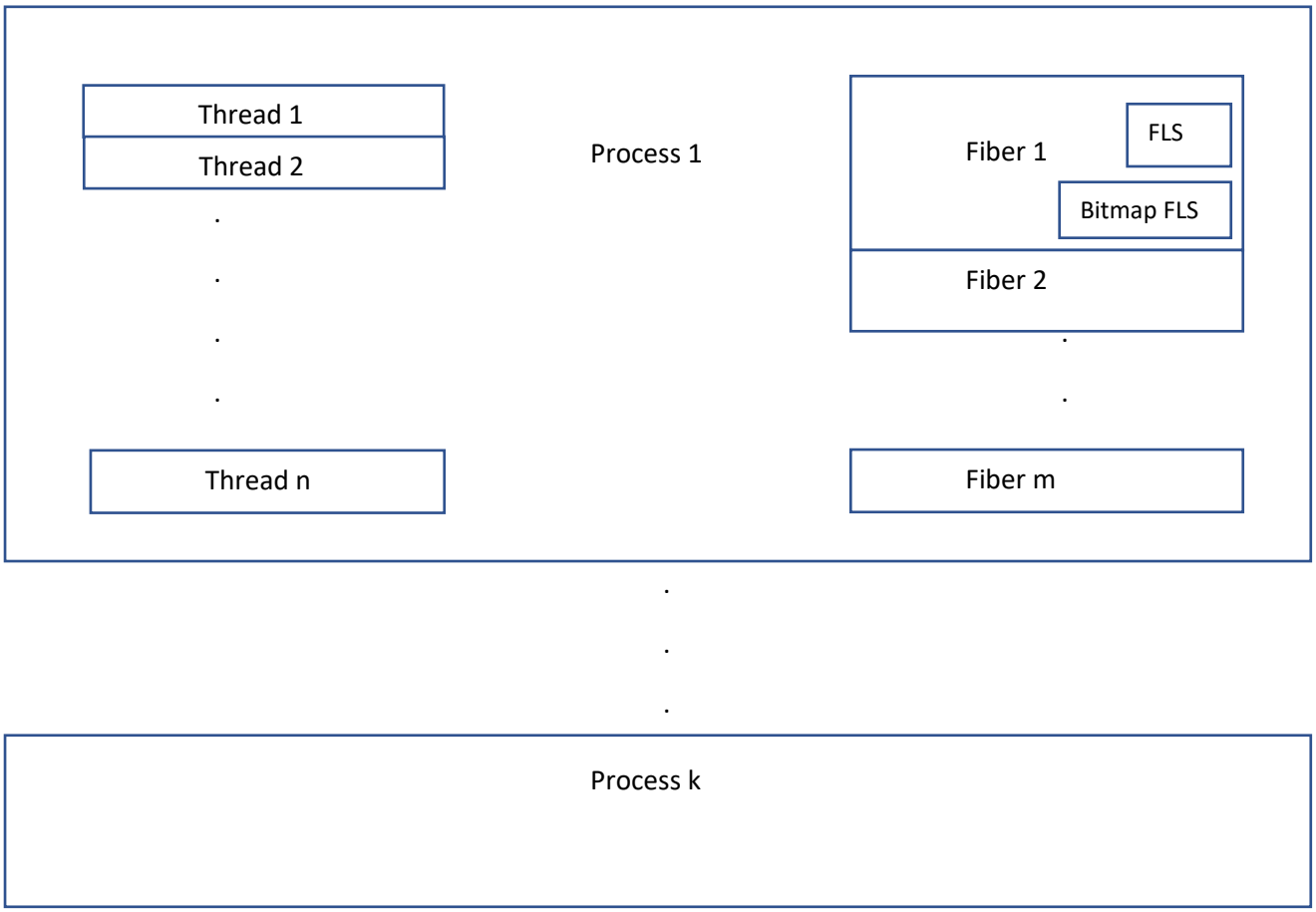


Fig 1. Design schema

We have designed the kernel level module following this schema. For each process we have a list of threads and a list of fibers. Each fiber has a fiber local storage (FLS) and a bitmap of the fiber local storage. The FLS is an array that acts exactly the same as thread local storage. With thread local storage (TLS), you can provide unique data for each thread that the process can access using a global index. One thread allocates the index, which can be used by the other threads to retrieve the unique data associated with the index.
The allocations are managed by the bitmap of the FLS, that is an array that contains for each cell a bit, 0 or 1 in case of allocation. The index of each cell in the bitmap is mapped with the index of each entry in the fiber local storage.
We have decided to implement the lists of processes, fibers and thread like hashtables, for obtaining optimal performance.
The number of the fibers is greater than the number of the threads.

# 3. Kernel Level Implementation

The data structures that we have used following the design schema are:

- struct process;
- struct thread;
- struct fiber.

The struct process identifies a process. It is initialized at the open of the device file in the main thread, in the *fib_open()* function.
The fields of this data structure are:

- struct hlist_node node, that represents the node of the hashtable of the processes;
- struct Fiber_Stuff fiber_stuff, the struct that contains fiber_base_stuff array for managing the static buffer "tgid_base_stuff" for the kprobe's handlers;
- pid_t last_fib_id, the last fiber's id allocated in the hashtable of the fibers;
- pid_t id, the id of the process.

```
struct Process{
    struct hlist_node node;
    DECLARE_HASHTABLE(listathread, 10);
    DECLARE_HASHTABLE(listafiber, 10);
    struct Fiber_Stuff fiber_stuff;
    pid_t last_fib_id;
    pid_t id;
};
```

Fig. 2

The struct thread is unique for each thread. It is initialized at the open of the device file in the main thread, in the *fib_open()* function.
The attributes of this data structure are:

- struct hlist_node node, that represents the node of the hashtable of the threads;
- struct Fiber *runner, that is the pointer at the fiber that is running;
- pid_t id, the id of the thread.

```
struct Thread {
        struct hlist_node node;
        struct Fiber *runner;
        pid_t id;

};
```

Fig. 3

The struct fiber represents a fiber and it is initialized in the function *do_fib_create()*. The fields of this data structure are:

- struct pt_regs* regs, that defines the way the registers are stored on the kernel stack during a system call or other kernel entry;
- struct fpu* fpu, the highest level per task FPU state data structure that contains the FPU register state plus various FPU state fields;
- void* entry_point, the execution entry point;
- struct Thread* selected_thread, the associated thread;
- spinlock_t lock_fiber, the lock that protects the fiber data structure in the critical sections;
- long long* fls, the pointer at the array of the fiber local storage;
- unsigned long *bitmap_fls, the pointer at the array of the FLS bitmap;
- unsigned long exec_time, the total executed time of a fiber;
- unsigned long last_activation_time;
- unsigned long correct_counter, the correct activations of a fiber;
- pid_t creator, the thread creator;
- pid_t id, the fiber's id;
- int failed_counter, the failed activations of a fiber.

```
struct Fiber{
    struct hlist_node node;
    struct pt_regs* regs;
    struct fpu* fpu;
    void* entry_point;
    struct Thread* selected_thread;
    spinlock_t lock_fiber;
    long long* fls;
    unsigned long *bitmap_fls;
    unsigned long exec_time;
    unsigned long last_activation_time;
    unsigned long correct_counter;
    pid_t creator;
    pid_t id;
    int failed_counter;
};
```

Fig. 4

We have used hashtables enabled with RCU (Read-Copy-Update) synchronization mechanism.

FUNCTIONS

IOCTL

At the kernel level the function that manages the ioctl (input/output control) on the device file is *fib_ioctl()*. This function switches depending on the command in input. For communicating with the user levels, we use *copy_from_user()*, that copies a block of data from user space, and *copy_to_user()*, that copies a block of data into user space.

CREATE FIBER

The *fib_create()* function creates a new fiber. In input the function takes the instruction pointer (void* func) the parameters (void* parameters), the stack pointer (void *stack_pointer), and the size of the stack (unsigned long stack_size). The function initialized the structure of the fiber with these parameters and adds the new fiber in the hashtable of the fibers. The fiber is not running and so it is not attached to a thread. In output the function returns a pointer to the structure fiber.

CONVERT THREAD TO FIBER

The *fib_convert()* creates a new thread if it is not present int the hashtable of the threads and makes it running. Then the function creates a new fiber and attaches the fiber to the running thread: in short words it creates the fibers and switches on it. At the end the function adds the thread and the fiber to the respective hashtables. In input the function doesn't take parameters and in output it returns the id of the new fiber.

SWITCH TO FIBER

The *fib_switch_to()* switches the execution context in the running thread from the running fiber to a different fiber. The function takes in input the id of the fiber that must be switched: if this id is equal to the running fiber the function fails. If the fiber that must be switched is running on another thread, the function fails.

FLS ALLOC

The *flsAlloc()* allocates a new index of the FLS in the bitmap. The bitmap is an array of values 0 or 1. Each allocation returns the last index of the cell with value 1.

FLS FREE

The *flsFree()* frees a FLS index in the bitmap. The function takes in input the index that must be deallocated. It set the cell at this index of the bitmap with 0 value.

FLS GET VALUE

The *flsGetValue()* gets the value associated with a FLS index. This value is saved in the buffer of the fiber local storage. The function takes in input the index of the cell containing the requested value.

FLS SET VALUE

The *flsSetValue()* sets the value associated with a FLS index. The function takes in input the index of the cell where setting the value in the FLS buffer, and the value.

PROC FILESYSTEM

The proc is a virtual filesystem used to access information about processes provided by the kernel. The filesystem is usually mounted in the /proc directory. It doesn't take up space on the hard disk and a limited amount of memory.
In the /proc/{pid} we have created a subfolder "fibers". For creating this folder we have used kprobes mechanism. A kernel probe is a set of handlers placed on a certain instruction address. A KProbe is defined by a pre-handler and a post-handler. When a KProbe is installed at a particular instruction and that instruction is executed, the pre-handler is executed just before the execution of the probed instruction. Similarly, the post-handler is executed just after the execution of the probed instruction.
We initialized the kprobe structures in the *fib_driver_init()*: we sets the post-handler, the pre-handler and symbol name of the original kernel function that must be managed. Then we register the kprobe structures. In the *fib_driver_init()* we unregister the kprobe structures.
The kernel functions that we manage are: *proc_pident_readdir()* and *proc_pident_lookup()*. We call the *Pre_Handler_Readdir()* function to modify the static tgid_base_stuff array that contains all elements in proc/{pid}. We add to this array the information of the "fibers" subfolder; the *Pre_Handler_Lookup()* does the same.
Then we have also used these functions for creating the attribute files: we create an array similar to tgid_base_stuff where each pid_entry has the information of the attribute file of the fiber.

# 4. User Level Implementation

IOCTL

The *ioctl()* function communicates at the kernel level with the device identified by the file descriptor fd and sends it a request command.
The third parameter (generally a void * or a char *) is not defined a priori by the prototype being indicated as '...'. The presence, the size in bytes and the verse (if it is input or output) strictly depends on the request command.
In the Fiber_interface.h we define the ioctl commands, with the request command, the size (in general we pass struct fiber_arguments) and the verse (write, read or nothing).

FIBER LIBRARY

In the Fiber_interface.c we have two function *__attribute__((constructor)) void fiber_init()* and *__attribute__((destructor)) void close_fiberlib()*. *__attribute__((constructor))* causes the function it is associated with to be called automatically before the *main()* function is entered. *__attribute__((destructor))* causes the function it is associated with to be called automatically after *main()* is entered. This mechanism is done for opening the char device in *fiber_init()* before the execution of the *main()* in the benchmark, and for closing the device in *close_fiberlib()* after the main.

# 5. Performance

We have analysed the performance of the software using both the benchmark and the command “time” on the shell.
The results with the benchmark shows that the initialization of a single fiber grows augmenting the number of the fibers. After 500 fibers there is a peak of the initialization time. This result depends from the fact that the creation of new fibers is done in the main thread, in a mono thread context.

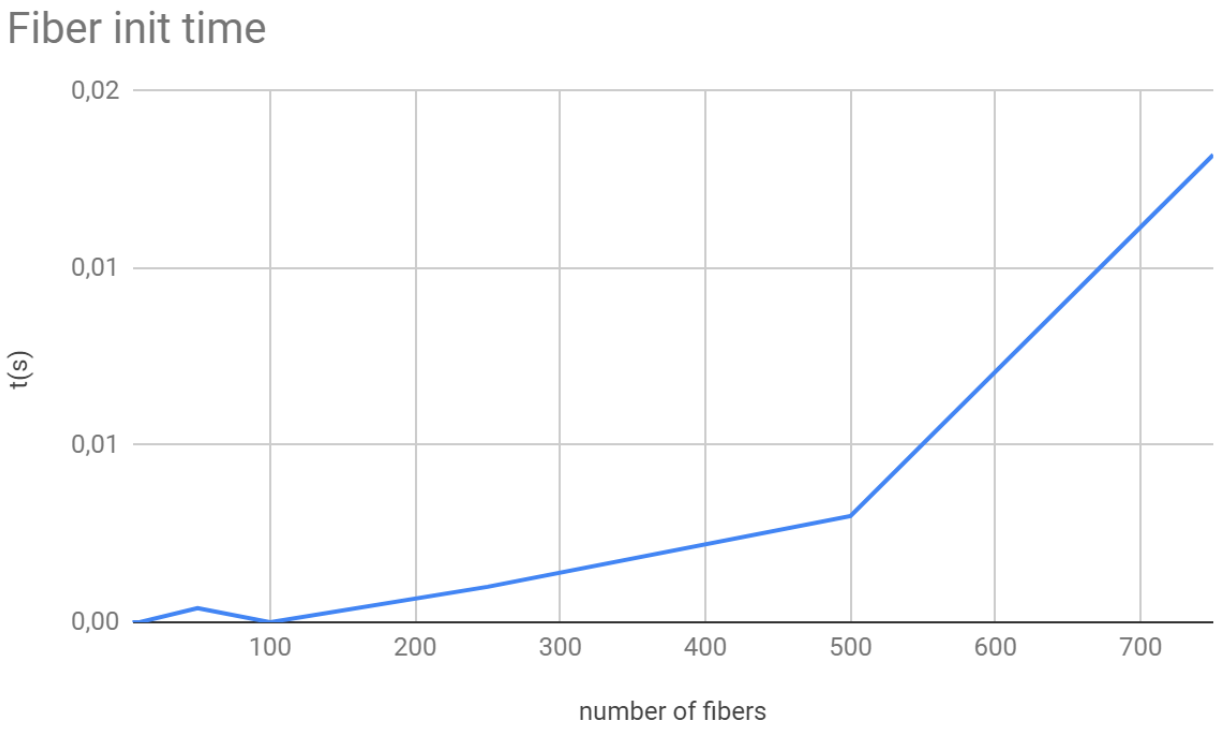


Fig. 5

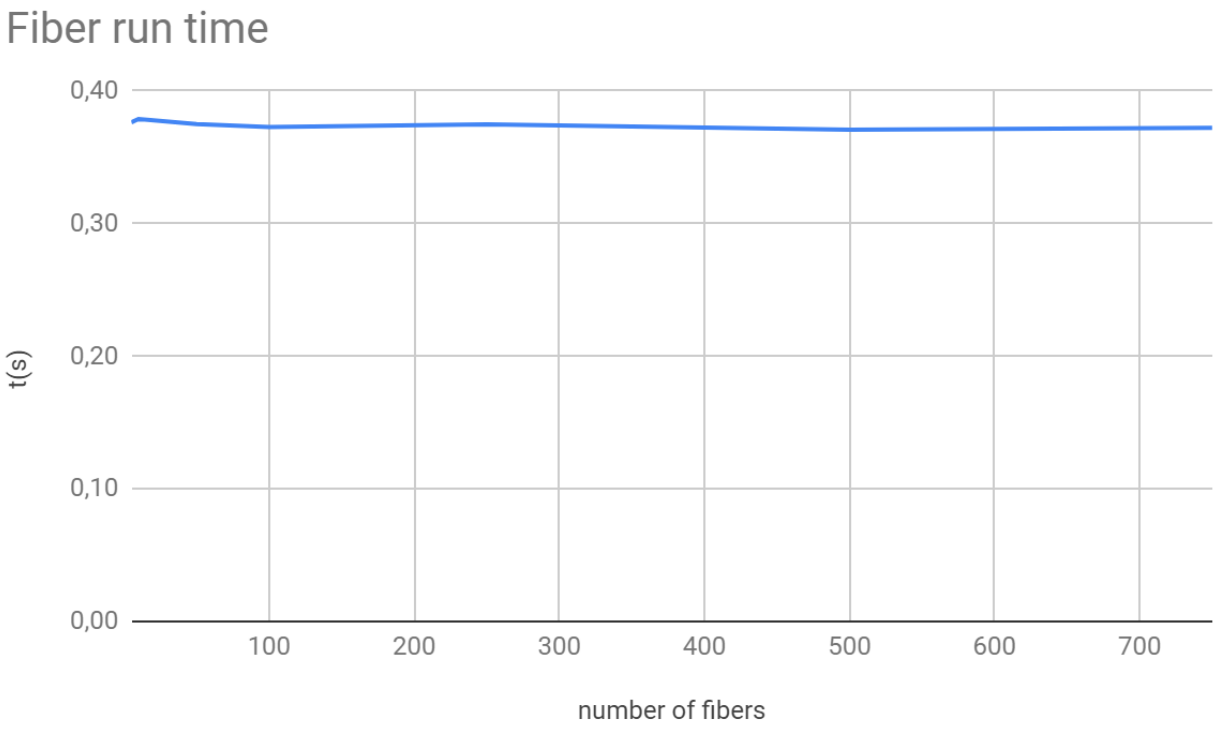


Fig. 6

This graph in figure 6 represents the relation between the running time of the single fiber and the number of the fibers. We have obtained a flat curve, due to the implementation with the hashtables. The hashtables in the best case has O(1) time complexity and if too many elements were hashed into the same key the worst case has O(n) time.

Then we have compared the performance of the kernel-level fibers with the user-level threads (ULT).
In the figure 7 we show the relation between the number of the fibers and the user-level total execution time.
In the figure 8 we have the relation between the number of the fibers and the kernel-level total execution time.
The average of the total user-level time is 121,40 seconds and the average of the kernel-level time is 6,20 seconds corresponding to an average number of 237 fibers: hence we can observe that the mean of the user time is much greater of the kernel time. At kernel level we have an average time of 6 seconds with an average number of 237 fibers and this result depends a lot by the user level implementation.

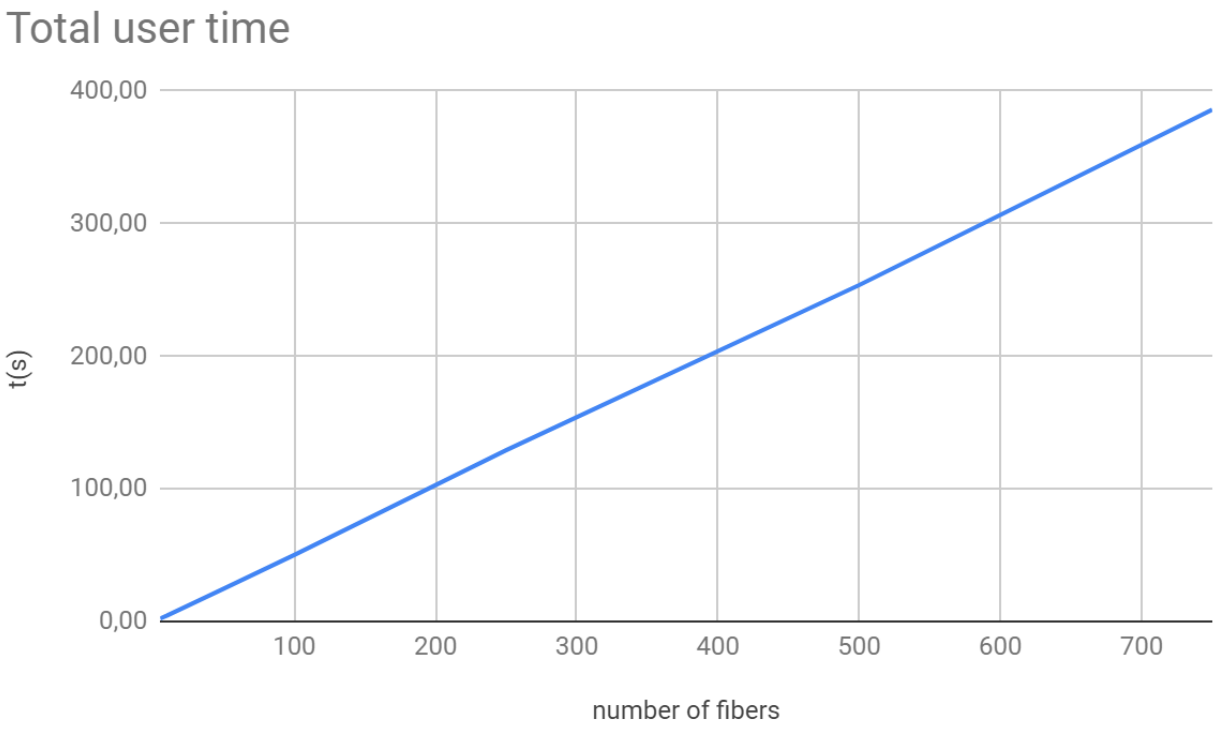
Total user time
400,00
300,00
200,00
100,00
0,00
t(s)
100
200
300
400
500
600
700
number of fibers

Fig. 7

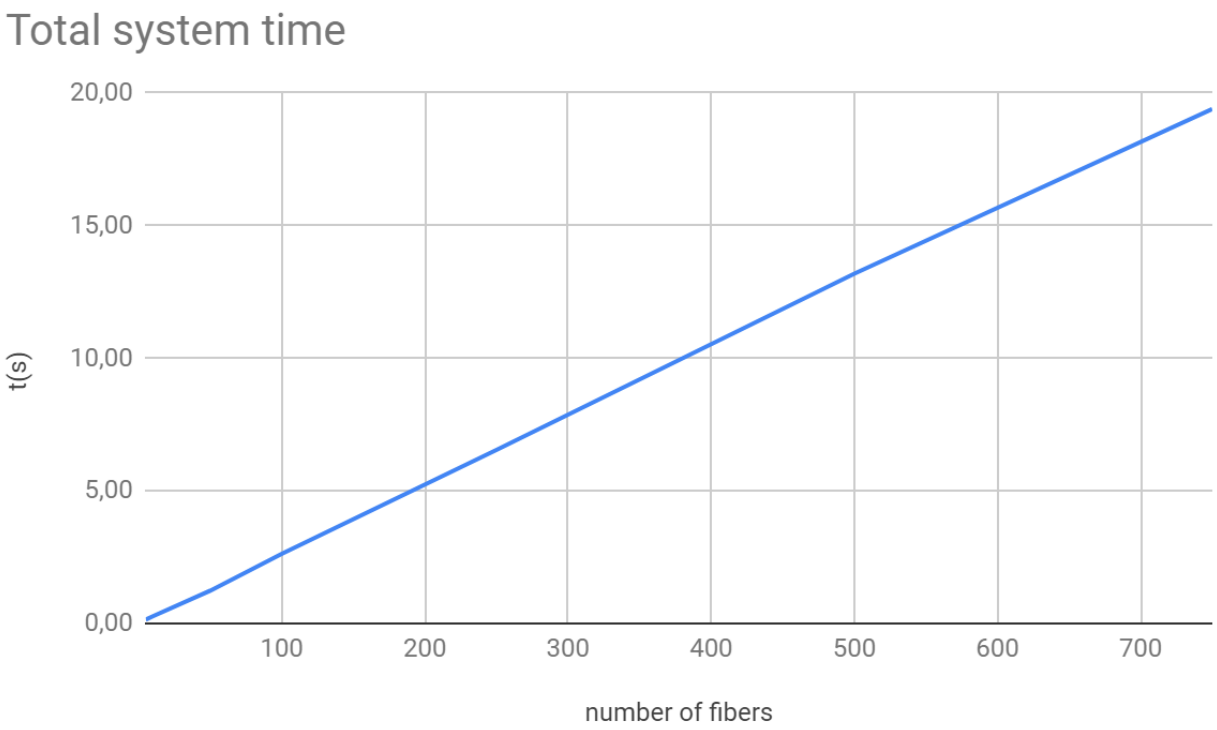
Total system time
20,00
15,00
10,00
5,00
0,00
t(s)
100
200
300
400
500
600
700
number of fibers

Fig. 8

# 6.Links and Resources

- Github repository: https://github.com/dariolitardi/linux_fibers
- https://elixir.bootlin.com/linux/v4.9/source/kernel
- https://lwn.net/