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 the fibers, made as final project for the *Advanced operating systems and virtualization* course. Fibers corresponds to User-Level Threads, but implemented at kernel-level in Windows operating systems. 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.

Fibers can be seen as a lightweight thread of execution, but, unlike threads that are scheduled by the kernel itself, they switch the execution from one fiber to another explicitly and the changes in the execution context are made by the newly implemented module.

MODULE

The module implemented in `module.c` is licensed under the *General Public License* and contains the main functions for initialization and exit.

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 added to have a sound and complete implementation of the fibers.

At the unloading of the module, there is a cleanup function that takes care of unregistering all kprobes and fiber_device.

DEVICE

Inside `device.c` we registered a new character device associated to the first available major number. We needed to create a class and the device in order to make it visible to the user and accessible by every user, not just by *sudo*.

The content of the device accessible through a *read* is stored inside the extern variable `string_message` and copied to the user. `string_message` is filled inside the file `ioctl.c` with all the new seven functions that will then be possible to call.

2 Kernel Level

We developed the whole logic of the program at kernel level, to make accessible to the user the minimal amount of information.

DATA STRUCTURES

We use three main data structure to handle every needed variable. The `struct process` is single inside the process and initialized by the first thread that converts into a fiber: we register here the number of active threads of the process, the number of fibers created until that point, two hashtable 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.

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 process and a pointer, that can be NULL, to the fiber that it is currently running.

Finally, the fundamental data structure is the `struct fiber`, one for each created fiber. Besides the information needed for the hashtable, there is a lock to let the threads choose a fiber to switch to in an atomic way. It is connected to the parent process and the thread that will execute it through a couple of pointers. It contains `struct pt_regs` and `struct fpu` to perform the switch and be able to restore this fiber's execution context in the right way. Here are saved the starting address of the stack of this particular fiber and its size. To handle the FLS, we need an array to put values, one in each cell, and a bitmap to quickly find available cells inside the array. The last fields in this data structure are needed to fill a file to be shown inside the proc subsystem.

IOCTL

The file `ioctl.c` comes into play each time is issued an ioctl call at user level. We have seven macros, one for each function, that are defined inside `ioctl.h` as **_IO** and are associated each one to a different number, from 0 to 6.

The main function is `fibers_ioctl()` where there are a series of *if* to distinguish exactly the call made according to the macro used. If the user was supposed to pass some parameters wrapped around a struct, there are a couple of checks on this structure and it is copied inside a local structure. After these precautions, we call the real function that performs the actual operations requested by the user.

In case of a switch, we pay attention to pick up 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 a multhithread context and only one thread at a time should perform the switch 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.

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 `convert_thread_to_fiber` is called by each thread that wants to use fibers and the very first thread that calls it, has to initialize an additional data structure related to the process the threads live in.

`convert_thread_to_fiber` will initialize all these structures, being aware of creating the `struct process` only if it doesn't yet exists, using a spin_lock.

CREATE_FIBER

SWITCH_TO_FIBER

FIBER_LOCAL_STORAGE

KPROBES

3 User Level

IOCTL Interaction

FIBER_LIBRARY

4 Proc subsystem

5 Performances