Openfibers Kernel Module Description

Advanced Operating Systems - Sapienza

Pietro Borrello

September 17, 2018

Contents

1 Introduction

In this paper we are going to present a brief description of a from scratch implemented Linux Kernel Module, which aims to give support to userspace processes to Fibers utilities. A fiber is a unit of execution of a process that must be manually scheduled by the application [1]. Each fiber runs in the thread that scheduled it, and any thread of a process can schedule any fibers owned by such process. A fibers that runs in a thread, runs in top of its contexts until that thread manually switches to any other fiber. When a thread running fibers is preempted, its currently running fiber is preempted. The same fiber resumes running when its thread resumes. Each fiber has the possibility to create and manage a Fiber Local Storage, in which it can use transparently a unique copy of a defined variable

2 APIs

The system has to offer some APIs to userspace processes, to let them ask the kernel to create and manage fibers.

- **ConvertThreadToFiber**: creates a fiber in the current thread. The calling thread is now the currently executing fiber.
- **CreateFiber**: create a new fiber from an existing fiber, being able to specify the stack size, the starting address, and the data the fiber will start from
- **SwitchToFiber**: execute any fiber created with CreateFiber or switch to a previously running fiber
- **FlsAlloc**, **FlsFree**, **FlsGetValue**, **FlsSetValue**: to manage the fiber local storage of the current fiber

3 Module Design Goals

The Fiber support will be served by the Linux Kernel Module exposing a character device in `/dev/openfibers` to communicate with, in order to be able to ask the kernel for fibers operations from userspace. The module will manage each fiber process context independently to favor isolation between all the processes, so that any process will not compete with the others for fibers resources. Moreover isolating the fibers for each process doesn't increase the attack surface for any user using fibers: with all the fibers in

the same structure any bug in the code would allow any process to corrupt sensible structure of any other (i.e. a root process).

The module is expected to work in an SMP system, and to favor the coexistence of multiple processes, and multiple thread, running at the same time, and with multiple fibers active each. Accesses to shared structures should be as quick as possible, possibly avoiding locks.

The module should expose information about active or running fibers to userspace, including the starting function, the number of successful and failed activation, the owner and the time of living. This should be ideally done exposing a `fibers` folder under `/proc/<pid>/fibers` containing a file for each fiber.

3.1 Fibers Operation

The first thread to be converted to fibers, should implicit initialize the fiber context for all the successive thread conversion and switches, and should set its current fiber running. After a thread is converted to a fiber, it should be able to create new fibers, setting the starting point and the initial parameters. Whenever a thread switches to a different fiber, we should guarantee it will work, unless the target fiber is yet running. Any concurrent execution of the same fiber should be avoided. We predicted the fiber switch to be the most frequent operation in fiber context, so it is critical to design it to be efficient.

Any fiber switch in principle should be designed to be non blocking, since any delay in the switch, would negatively affect the most part of applications designed on top of fibers. We allow instead the kernel to rely on lock for safe initialization and safe cleanup, as they are expected to be less frequent, but critical to be done in isolation.

3.2 FLS Operation

We expect every fiber will have a highly dense usage of `FLS` facilities. Therefore we just keep it simple. Any thread should access the fiber local storage without any delay with respect to normal variable accesses (without accounting for the syscall delay needed to communicate with the kernel). Therefore we refused to accept any design that would introduce a non constant number of memory accesses for local storage operations.

3.3 Components

The Fiber implementation will have two components: A loadable linux kernel module that will implement the desired operations, at kernel level, to be as efficient as possible; and a shared library to be included in user-level processes to provide a user friendly interface with the kernel module. To favor information hiding, and to simplify interaction, the library should manage the formats and structures needed by kernel level APIs, exposing to the user a simple directly callable interface.

4 Module Implementation

The Linux Kernel Module managing fibers is implemented as a character device in `/dev/openfibers` exposing a single entry point to ask for operations through the `ioctl()` system call. The following commands to be issued to the syscall are declared: they reflect all the possible APIs operations that can be issued to the Fibers.

```
1 #define OPENFIBERS_IOCTL_CREATE_FIBER [...]
2 #define OPENFIBERS_IOCTL_SWITCH_TO_FIBER [...]
3 #define OPENFIBERS_IOCTL_CONVERT_TO_FIBER [...]
4 #define OPENFIBERS_IOCTL_FLS_ALLOC [...]
5 #define OPENFIBERS_IOCTL_FLS_FREE [...]
6 #define OPENFIBERS_IOCTL_FLS_SET [...]
7 #define OPENFIBERS_IOCTL_FLS_GET [...]
```

The module maintains process fiber information in a Red Black Tree [2]. The tree contains the set of fibers for each thread group, indexed by `tgid`. Any `tgid` node is declared as follows:

```
1  struct fibers_by_tgid_node
2  {
3      struct rb_node node;
4      pid_t tgid;
5      struct rb_root *fibers_root;
6      struct rw_semaphore fibers_root_rwsem;
7      struct kref refcount;
8      atomic_t max_fid;
9      struct proc_dir_entry *tgid_proc_fibers;
10 };
```

Any access to the fibers set is protected by a RW semaphore and a kref counter avoids premature deletion. Therefore each change in the structure of the tree (i.e. adding a new fiber) will be done in isolation, while each access to fibers will be done concurrently.

The set of fibers for each process is still represented by a Red Black Tree (an IDR though would equally have done the job) and each node contains a pointer to the fiber struct:

```
1  typedef struct
2  {
3      fid_t fid;
4      atomic_t running;
5      void (*start_address)(void *);
6      exec_context_t context;
7      unsigned long idx;
8      unsigned long fls[4096];
9      unsigned long fls_idx;
10     struct proc_dir_entry *proc_entry;
11     pid_t created_by;
12     unsigned long activations;
13     atomic_long_t failed_activations;
14     struct timespec total_ts;
15     struct timespec tmp_ts;
16 } fiber_t;
```

It contains all the information needed to present statistics to the user, and to manage fiber operations. An `atomic_t` variable protects a running fiber from being accessed in the fibers switch through a `cmpxchg` operation. Each fiber maintains its context of execution that is updated every times a thread leaves it and its declared as:

```
1  typedef struct __exec_context_t
2  {
3      unsigned long long orig_rax;
4      unsigned long long rax;
5      unsigned long long rdx;
6      unsigned long long rcx;
7      unsigned long long rsp;
8      unsigned long long rsi;
9      unsigned long long rdi;
10     unsigned long long r8;
11     unsigned long long r9;
```

```
12      unsigned long long r10;
13      unsigned long long r11;
14      unsigned long long rip;
15      unsigned long long flags;
16      struct fpu fpu_context;
17  } exec_context_t;
```

It maintains all the registers that are saved by the kernel upon the `ioctl` syscall along with the fpu context. The other registers are saved and restored by the userspace shared library.

4.1 Fibers Operation

All the fibers operation are implemented by the `LKM` with the help of the shared library.

ConvertToFiber Any thread calling the `ConvertToFiber` library function results in opening its own file descriptor for `/dev/openfibers`, which will be the entry point for all its operations. Then the respective `ioctl` call instructs the kernel to create the proper structures for the current thread that will be converted to a fiber. Opening `/dev/openfibers` will result in the initialization of the tgid structure for the current thread group, if any other thread hasn't done it yet. Such operation is serialized by a dedicated mutex to prevent concurrent initializations of the same structure. The last release of the file (by the kref counter) will cleanup the structure.

The kernel then creates a fiber for the current running thread and inserts it in the dedicated struct. The current fiber is saved by the kernel in the `private_data` field of the `struct file` passed to `ioctl` call. This is why each thread needs a dedicated file descriptor.

CreateFiber Creating a new fibers requires a new stack. The shared library allocates the needed memory to serve the request for the stack and passes the pointer to the `ioctl` call. Then the kernel module creates the needed structures to hold fiber information, setting the start address (i.e. saved `rip`), the start parameter (i.e. saved `rdi`) and the new stack pointer to the requested values. Thus when switching to the newly created fiber the values will be moved to the registers. Finally the fiber ID assigned to the new fiber is returned.

SwitchToFiber While switching to the new fiber, current register values should be saved. The shared library saves callee-preserved registers which aren't saved on kernel entry, while the kernel module saves all the others. Current fiber is taken from the `private_data` field of the `struct file` passed to `ioctl`. Moreover the kernel module saves the fpu context using `fpu__save`. The switch to the next fiber is implemented by a `cmpxchg` operation on the `running` field: if it fails it means the next fiber is yet running, otherwise the registers saved in the net fiber context are restored. Finally the old fiber is released setting it `running` field to 0.

The whole operation is non blocking and the complexity is `O(log(n))` with `n` fibers for the current `tgid`, due to the access to the Red Black tree (whose root is cached in all the nodes) to search the next fiber by ID.

4.2 FLS Operation

Fiber Local Storage operation should not introduce any delay in the storage access, as if a local variable is accessed. Therefore all the operations are implemented through accesses to the kernel level array maintained for each fiber. The overall cost for each operation is `O(1)`. The drawback is that every fiber is not allowed to have more than a fixed number of FLS variables (set to 4096 by default).

4.3 Proc Statistics

While developing the software, exposing info in `/proc/<pid>/fibers/<fid>` was one of the most challenging tasks. The linux kernel doesn't expose any facility to add entries to the `/proc/<pid>` folder: they are statically determined at compile time. Any time a process accesses `/proc/<pid>` the entries are dynamically built. The functions `proc_pident_readdir` and `proc_pident_lookup` are respectively used to list the contents of `/proc/<pid>` and to access a file in it. They are accessed through `struct inode_operations proc_tgid_base_inode_operations` and `struct file_operations proc_tgid_base_operations` by the `proc` subsystem.

We hooked the function pointers in `struct inode_operations proc_tgid_base_inode_operations` and `struct file_operations proc_tgid_base_operations` to point to some self defined wrappers to functions that would simply add an entry to the returned values, to include a symbolic link to the folder `/proc/fibers/<pid>` which actually contains the fiber stats indexed by `/proc/fibers/<pid>/<fid>`.

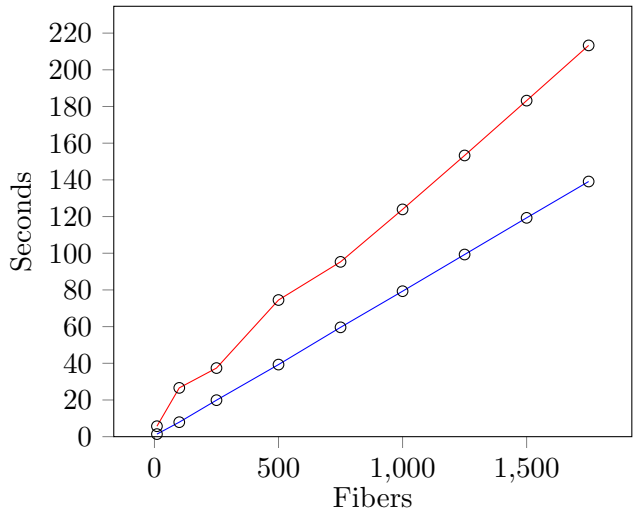
Therefore we managed to have for example:

```
$ cat /proc/32706/fibers/52
   running = 0
   start = 0x402740
   created by = 32706
   activations = 2
   failed activations = 0
   total time = 0.539752864
```

5 Evaluation

The module has been tested from linux kernel 4.4 to linux kernel 4.17. The linux kernel module is evaluated against a standard userspace fiber implementation based on sigaltstack. All the times are referred to runs against a single process using fibers to implement a simple telephone calls simulation on a Intel i7 2nd generation Quad Core, running Ubuntu 16.04 with linux kernel 4.4.

The scatterploit displays in blue the time in seconds for running the simulation with `n` fibers using the linux kernel module, and in red the time for running it with the userspace implementation. All the times were reported as the average of 5 successive execution of the same task to reduce entropy.



From the plot we can infer a speedup with respect to the userspace implementation ranging from `4x`, with very few fibers (ten or twenty), to still more than `1.5x` with more than a thousand of fibers.

References

[1] Microsoft, `docs.microsoft.com/en-us/windows/desktop/procthread/fibers`, 2018.

[2] Kernel Docs, lwn.net/Articles/218239/