

Advanced Operating Systems and Virtualization

Project a.a. 2017/2018 - Fibers

Carmine Maria Mansueto - 1646454

June 2, 2019

1 Main Data Structures

- **struct process_t**: data structure representing a whole process (thread group), storing the TGID, the number of its converted threads and its created fibers, an atomic counter in order to give **ids** to the fibers (FID - Fiber ID) and two hashtables, one for its fibers and one for its converted threads. Thanks to the per-process atomic counter, two processes can give the same FID to their fibers. This choice has been preferred w.r.t. a global counter in order to avoid the problem of the re-use of the FIDs once fibers get removed (an implementation using bitmasks would have been needed).
- **struct fiber_context_t**: a fiber data structure keeps the FID, the structure to hold general-purpose registers context, the structure to hold FPU context, a variable "pid_t thread" which identifies the PID of the thread running the fiber (0 if the fiber is free, -1 if the fiber is no longer accessible). Moreover, there is a lock for handling the concurrency in case of "switch_to", a structure for the *Fiber Local Storage* (FLS), and some field to be exposed in PROC.
- **struct thread_t**: a thread data structure just holds the thread PID, a pointer to the "struct process_t" of its Thread Group and a pointer to the "struct fiber_context_t" of the fiber the thread is executing. It has been chosen to keep the pointers to the "process_t" and "fiber_context_t" rather than the TGID and FID in order to avoid to go looking for in the hashtables, so for efficiency reasons.

2 Kernel Code Structure

The kernel code has been implemented into a **Loadable Kernel Module** rather than extending the source code of the Linux Kernel. The module registers a device "/dev/fibers" into the system and associates to it some **ioctls** by means of which it is possible to interact with the module, simulating the system calls.

The module is divided into two C files, together with the corresponding two headers files. The "fiber_module.c" one is just the code that sets up the module, the **ioctls** and some **kprobes**, while the most interesting one is the "fiber_utils.c" one which I'm going to briefly describe, and that contains the core functions which perform the core work to implement the fibers.

A global hashtable of 1024 buckets stores the "struct process_t" structures of all the running processes which have started to "play" with fibers. This structure is protected by a lock when two or more threads of the same process concurrently call "convert_thread()" (otherwise there is the risk that the threads create more than one "struct process_t" structures which refer to the same process).

The first thread which calls "convert_thread()" creates the "struct process_t" and inserts it in the global hashtable. Such struct keeps two hashtables, each of which has again 1024 buckets, one for its converted threads and one for its

fibers. Each hashtable (including the global one) handles the concurrency by means of the RCU operations of the kernel "linux/hashtable.h" library, only the global hashtable is further protected by a lock for the reasons just described.

So, in the "convert_thread()" function it is created, if not already existing, the process structure, and then the one of the current thread and the one of the fiber to which the current thread will switch. Indeed, the fiber just created gets immediately marked as "occupied" by the current thread.

The "create_fiber()" instead does nothing more than creating a fiber structure and putting inside it the handler, which has been passed from userspace, which will be started to execute by the first thread which will switch to the fiber. The fiber is immediately marked as "free" and so ready to be executed.

The "switch_to()" takes the target fiber and checks whether it exists or not, if it exists it tries to see if some other thread is currently trying to steal the target fiber, if not it checks whether the fiber is free (if target fiber->thread == 0), if yes the fiber is free and the actual "context switch" happens, saving the CPU state of the current thread into the old fiber (the one the thread was executing, if any) and storing into the CPU the state of the target fiber.

For what concerns the FLS, each fiber has an array of *long long* of size 1024, and a bitmask handles the free entries, the implementation has been done using the utilities of the kernel libraries ("find_first_zero_bit()", "clear_bit()" and so on).

The "fls_alloc()" creates the array and the bitmask if not already existing, and finds the first free entry by means of "find_first_zero_bit()" passing the bitmask as parameter, then it occupies the entry with "set_bit()" passing as parameter the index of the entry which has been returned by the "find_first_zero_bit()".

The "fls_free()" does exactly the opposite, checks whether the fls array or the bitmask are not NULL (if yes it fails) and calls the "clear_bit()" passing as parameter the index of the target entry to be freed". The "fls_get()" performs the same checks of the "fls_free()" and also checks if the index passed corresponds to an allocated entry by means of the "test_bit()" passing as parameter the index and the bitmask.

The "fls_set()" performs the same checks of the "fls_get()" and in case of success it stores the passed value in the specified entry.

3 Data Structures Cleanup

The cleanup of the stuff allocated by the module is delegated to the "doexit_entry_handler()" function. As the name suggests, this function is the handler of the kprobed "do_exit" function, which is the kernel function called when a single thread gets killed (either singularly after a "pthread_exit()" or when a whole Thread Group gets taken down after an "exit()" call). Notice that this is an entry handler, so it is called before the actual "do_exit()" is invoked, if it is called after it would be too late since informations about the current thread will be lost.

This function checks whether the "dying" thread is the last one of the Thread Group, if not only the current thread and the associated fiber (if any) have to be removed, the fiber is marked as "unavailable" (fiber->thread = -1) rather than destroying its data structure, then the thread data structure is destroyed.

If instead the thread which is "dying" is the last one of the Thread Group, a total cleanup is performed. In particular, all the fibers data structures and the threads data structures get destroyed (together with the hashtables of the current process), and in the end the data structure of the process is destroyed, removing it from the global hashtable. Notice that when a thread "dies" it is because either it has finished its routine or because it generated an error, in both the cases it is correct to destroy the fiber the thread was executing (if any).

4 Proc Filesystem

The functions that perform the main work in PROC PID are the "proc_pident_readdir" and the "proc_pident_lookup" ones, so they have been kprobed in order to intercept them and to modify the content of the /proc/PID directory of the target process. In particular, when /proc/PID is accessed and the target process has a pool of fibers, the directory "fibers" gets showed, this thanks to the "kprobe_proc_readdir_handler". The directory "fibers", which is nothing more than a **pid_entry**, gets instantiated by means of the DIR macro (view proc/base.c in the Kernel source code). The "kprobe_proc_lookup_handler" performs the same things when /proc/PID gets surfed.

Once in "/proc/PID/fibers", when the directory content gets listed the kretprobe intercepts the call and executes the "fibdir_readdir", which is in charge of taking the PID of the process and showing its pool of fibers, instantiating in "/proc/PID/fibers" a pseudofile for each fiber in its pool.

Each of these pseudofiles has as functions the ones specified in "struct fibentry_fops", so they are the "fibentry_read" which gets called when the pseudofile is read.

The "fibentry_read" takes the FID from the name of the pseudofile read, looks for the fiber structure in the hashtable of fibers of the proper process, and prints to the screen the informations contained into the PROC fields of the fiber structure such as the state (RUNNING or WAITING), the thread executing it (if any), the thread which created it, execution time of the fiber and so on.

5 Benchmark Integration

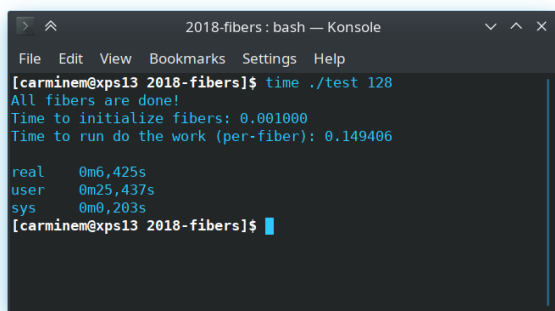
Taking the benchmark code, the file "fiber.h" has been modified, basically the "#define USERSPACE" has been removed so that the code will use the functions defined below the "#else" statement, so it has been sufficient to declare those functions with the names of the ones of the userspace library.

Finally, in the available Makefile it has been added, to the SRC variable which contains all the C files to be compiled, the file "fiber.library.c" that is the implementation of the userspace library.

6 Benchmark comparison Userspace vs Kernelspace

The following tests have been performed on a Dell XPS13 laptop, equipped with a 7th gen. Intel Core i5-7200U CPU, dual core with Hyperthreading, 2.5 GHz as base frequency, up to 3.1 GHz on TurboBoost, the distro is Manjaro Linux (Arch Linux based distro) running the kernel **4.19.45-1-MANJARO**.

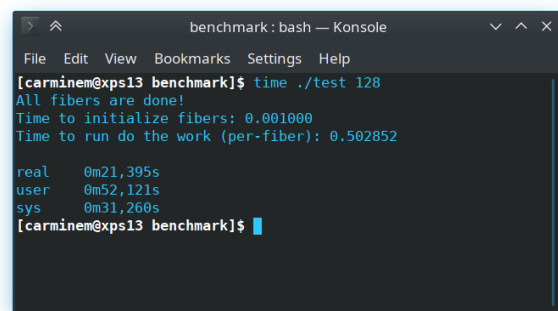
Note: in order to measure the actual time spent processing, all the *printf* functions have been removed in users code and all the *printk* functions have been removed from kernel code.



```
2018-fibers: bash — Konsole
File Edit View Bookmarks Settings Help
[carminem@xps13 2018-fibers]$ time ./test 128
All fibers are done!
Time to initialize fibers: 0.001000
Time to run do the work (per-fiber): 0.149406

real    0m6.425s
user    0m25.437s
sys     0m0.203s
[carminem@xps13 2018-fibers]$
```

(a) userspace 128 fibers



```
benchmark: bash — Konsole
File Edit View Bookmarks Settings Help
[carminem@xps13 benchmark]$ time ./test 128
All fibers are done!
Time to initialize fibers: 0.001000
Time to run do the work (per-fiber): 0.502852

real    0m21.395s
user    0m52.121s
sys     0m31.260s
[carminem@xps13 benchmark]$
```

(b) kernelspace 128 fibers

```
2018-fibers : bash — Konsole
File Edit View Bookmarks Settings Help
[carminem@xps13 2018-fibers]$ time ./test 256
All fibers are done!
Time to initialize fibers: 0.002000
Time to run do the work (per-fiber): 0.149047

real    0m12,705s
user    0m50,539s
sys     0m0,163s
[carminem@xps13 2018-fibers]$
```

(a) userspace 256 fibers

```
benchmark : bash — Konsole
File Edit View Bookmarks Settings Help
[carminem@xps13 benchmark]$ time ./test 256
All fibers are done!
Time to initialize fibers: 0.002000
Time to run do the work (per-fiber): 0.502547

real    0m42,774s
user    1m44,711s
sys     1m1,789s
[carminem@xps13 benchmark]$
```

(b) kernelspace 256 fibers

```
2018-fibers : bash — Konsole
File Edit View Bookmarks Settings Help
[carminem@xps13 2018-fibers]$ time ./test 512
All fibers are done!
Time to initialize fibers: 0.006000
Time to run do the work (per-fiber): 0.149438

real    0m25,566s
user    1m41,616s
sys     0m0,333s
[carminem@xps13 2018-fibers]$
```

(a) userspace 512 fibers

```
benchmark : bash — Konsole
File Edit View Bookmarks Settings Help
[carminem@xps13 benchmark]$ time ./test 512
All fibers are done!
Time to initialize fibers: 0.003000
Time to run do the work (per-fiber): 0.503150

real    1m25,894s
user    3m30,885s
sys     2m4,431s
[carminem@xps13 benchmark]$
```

(b) kernelspace 512 fibers

```
2018-fibers : bash — Konsole
File Edit View Bookmarks Settings Help
[carminem@xps13 2018-fibers]$ time ./test 1024
All fibers are done!
Time to initialize fibers: 0.012000
Time to run do the work (per-fiber): 0.149607

real    0m51,073s
user    3m23,388s
sys     0m0,493s
[carminem@xps13 2018-fibers]$
```

(a) userspace 1024 fibers

```
benchmark : bash — Konsole
File Edit View Bookmarks Settings Help
[carminem@xps13 benchmark]$ time ./test 1024
All fibers are done!
Time to initialize fibers: 0.007000
Time to run do the work (per-fiber): 0.506257

real    2m52,806s
user    7m3,549s
sys     4m10,749s
[carminem@xps13 benchmark]$
```

(b) kernelspace 1024 fibers

In order to make a comparison, it is useful to see, for what concerns the kernel space version, the **sys** time which tells us the time spent by the module. In all the four runs it can be noticed that the userspace version is slightly more efficient than the kernelspace one, in particular in the run of 1024 fibers the userspace was in advantage of more or less 47secs. For what concerns the time to initialize the fibers the kernelspace version is slightly more efficient, but the converse happens for the time of the per-fiber work, for which the userspace version is 3-4 times faster.

References

- [1] "Windows Fibers documentation",
"<https://docs.microsoft.com/en-us/windows/desktop/procthread/fibers>"
- [2] "Stack Frame Layout on x86-64",
"<https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64/>"
- [3] "StackOverflow - Stack Alignment",
"<https://stackoverflow.com/questions/26866723/main-and-stack-alignment>"
- [4] "Linux Kernel Code (Bootlin) - Kprobes",
"<https://elixir.bootlin.com/linux/v4.19.20/source/include/linux/kprobes.h#L73>"
- [5] "Kretprobe example on GitHub",
"https://github.com/spotify/linux/blob/master/samples/kprobes/kretprobe_example.c"
- [6] "Linux Kernel Code (Bootlin) - FPU Internals",
"<https://elixir.bootlin.com/linux/latest/source/arch/x86/include/asm/fpu/internal.h#L30>"
- [7] "Kernel Hashtables example on GitHub",
"https://github.com/JagadeeshPagadala/hash_table/blob/master/hash_table.c"
- [8] "Atomic Variables",
"<http://tuxthink.blogspot.com/2011/10/atomic-variables.html>"
- [9] "Luca Abeni - Kernel Locking",
"http://retis.sssup.it/~luca/KernelProgramming/kernel_locking.pdf"
- [10] "Task_Struct cpu time",
"<https://stackoverflow.com/questions/16726779/how-do-i-get-the-total-cpu-usage-of-an-application-from->
- [11] "PROC",
"<http://tldp.org/LDP/lkmpg/2.6/html/c708.html>",
"<http://tuxthink.blogspot.com/2012/01/creating-folder-under-proc-and-creating.html>",
"<https://elixir.bootlin.com/linux/v4.19.28/source/fs/proc/base.c#L2448>",
"<https://stackoverflow.com/questions/40431194/how-do-i-access-any-kernel-symbol-in-a-kernel-module>",
"<https://ops.tips/blog/how-is-proc-able-to-list-pids/#linux-and-its-pids>"