Unix Fibers
AOSV Final Project

Andrea Mastropietro, Umberto Mazziotta
Sapienza University of Rome

A.Y. 2018/19

1 Specifications

The aim of the project is the implementation of Windows Fibers in Linux Kernel. Fibers are the Windows kernel-level implementation of User-Level Threads. The assignment consisted into implementing the facilities related to the conversion of a Thread into a Fiber, the creation of new Fibers and the possibility to switch from a Fiber to another. Each Fiber has a Fiber Local Storage (FLS) which can be accessed by that Fiber only. Each Fiber has its own information exposed in the proc FS under the folder related to the process the Fiber belongs to. Fibers are implemented in the project using a Kernel module. Such module registers and creates a char device, with which the user-space library communicates using IOCTLs.
The following sections will cover the description of the implementation of the user-space library, the kernel module and the exposure to the proc FS. Finally, we will talk about the comparison between the user-space and our kernel-space implementation.

2 User-Space Library

The functions exposed to the user are `ConvertThreadToFiber`, `CreateFiber`, `SwitchToFiber` and four more functions employed to manage the FLS.
Besides `ConvertThreadToFiber` and `CreateFiber` all the functions are wrappers for IOCTL calls to the char device.
`ConvertThreadToFiber` is endued with opening a descriptor to the device for the calling process. The function ensures that if a child of a process that has already called the function calls the `ConvertThreadToFiber` itself, it is given a different descriptor. The function then calls the related IOCTL.
`CreateFiber` allocates the stack aligning it to the the dimension of a page. Since the kernel expects to find at the base of the stack the return address, we have to let stack pointer point 8 bytes below the top of the newly allocated stack.

3 Kernel-Space Code

The module uses a hashtable to keep track of the processes that are using Fibers. The hashtable size is chosen according to the default maximum number of possible process in the

system. In such hashtable we store information regarding the PID of the process, two queues for the Fibers (one for the active and one for the waiting ones) and the ID of the next Fiber to be allocated. The data structure has a lock that guarantees the safety of concurrent accesses to the entries of the table.

3.1 Fibers

Each Fiber is represented by a struct containing several fields, such as the fiber_id, the last thread the fiber run on, the cpu and fpu states, the FLS and all the fields that will be exposed in proc fs.
The `open` function checks if the current process is in the hashtable. If it is not present it is added to the hashtable and the function sets a pointer to the process in the private data field of the struct file. In this way any process can easily access the data structure representing itself without the need of scanning the hashtable.
The `release` function does nothing since all the cleanup work is done using a kprobe.

IOCTL According to the parameter passed to the `ioclt` function, we check if the buffer passed as argument is accessible. Then, the corresponding IOCTL is called.

- IOCTL_CONVERT: the function checks if the caller is a Fiber; if it is true the conversion fails, otherwise we allocate a new Fiber setting as entry point the current instruction pointer value.
- IOCTL_CREATE: it checks if the caller is a Fibers; if it is false it fails, otherwise we allocate a new Fiber and we add it to the queue related to the waiting Fibers in the process hashtable.
- IOCTL_SWITCH: the function checks if the caller is a Fiber; if true we look for the Fiber we want to switch to in the waiting queue. We then scan the active queue. If the Fiber we want to switch to is active, the switch fails. On the contrary, if such Fiber is waiting and the calling Fiber is in the active queue, we copy the context of the cpu and fpu of *current* into the struct representing the calling Fiber and the contexts of the Fiber we want to switch to are moved into *current*. The two Fibers involved in the switching are swapped from one queue to the other.

3.2 FLS

The Fiber Local Storage is implemented as an array of *long long* along with a bitmask. The value 1 in the position n of the mask means that the index is available for usage, 0 means the opposite. The size of such array is 1024 *long long* for each Fiber. The struct holding the FLS is created contextually with the creation of the Fiber. IOCTLs are used to work with the FLS.

- IOCTL_ALLOC: the function checks if the FLS array is available; if not, it is dynamically allocated. If the array is allocated, we check in the bitmask if there is a bit set to 0: it is set to 1 so that it can be used by the Fiber.

- IOCTL_SET and IOCTL_GET: both functions check if the bit corresponding to the index we want to access is set to 1. If it is true, the get function returns the content while the set function sets the content that was passed as argument. On the contrary, if the bit is set to 0, an error is returned.
- IOCTL_FREE: the function sets the bit corresponding to the index passed as argument to 0.

3.3 Cleanup

The cleanup is managed by registering a kprobe on the `do_exit()` function. We scan the hashtable to find the process which the thread that exited belongs to. Then, we scan the running queue to look for the Fiber running on the top of the current thread, we remove it from the queue and we free the memory. Consequently, we check if there are any running Fibers left; if there are we exit from the function, otherwise it means that the whole process terminated and so we delete all the Fibers in the waiting queue.

3.4 proc

Our approach in exposing Fibers information in the proc FS is done through the usage of kretprobes.
In order to show the *fibers* folder inside the <PID> folder in *proc* we probe the functions `proc_pident_lookup` and `proc_pident_readdir`. We use a pre-handler in order to change the parameters that the functions take as input; we allocate a new array, copy the content of the original input in it and add the pid_entry corresponding to the *fibers* folder. Then, we call the original functions that will work on the new input. Finally, we employ a post-handler to release the memory used for the new input. We associate to the new pid_entry a struct file_operations for which we defined a function, `fiber_readdir`, for the .iterated_shared operation. The latter is the function that creates the files related to each Fiber of the process by scanning both the running and the waiting queues. The file operations we defined rely on the original file operations; we call the original functions by passing as parameters the data structures we defined. We associate to the files a struct file_operations for which .read operation we define a function, `fiber_read`, that writes all the information we are interested in about the Fiber, such as whether it is running or not, the initial entry point, the parent thread, the number of successful and failed activations and the total execution time.
We retrieve the original function by using the `kallsyms_lookup_name` function.

4 Benchmark

The experiments were both run on a physical machine with a 8-core i7-4710HQ, 16GB of RAM mounting a Ubuntu 18.04 kernel 4.15 and on a virtual machine with a 4-core i7-7700HQ with 8GB of RAM mounting the same operating system and kernel version of the physical machine.

4.1 Virtual Machine

The following tables show the time elapsed for the initialization and the execution of the Fibers according to the number of fibers and the number of processes. Table 1 shows the

results for the user-space implementation while Table 2 for the kernel-space one.

	#Fibers	16	64	128	512	1024
Monoprocess	Init time (s)	0.0000000	0.0010000	0.0010000	0.0040000	0.0080000
	Running time (s)	0.2570630	0.2624060	0.2526880	0.2488140	0.2778080
Multiprocess (4)	Init time (avg s)	0.0035000	0.0105000	0.0097500	0.0167500	0.0200000
	Running time (avg s)	0.8582655	0.9955703	1.1404298	0.9817238	1.0038593

Table 1: User-space results.

	#Fibers	16	64	128	512	1024
Monoprocess	Init time (s)	0.0000000	0.0010000	0.0010000	0.0030000	0.0060000
	Running time (s)	0.301438	0.289719	0.293227	0.286602	0.288016
Multiprocess (4)	Init time (avg s)	0.00775	0.0065	0.0065	0.00775	0.01925
	Running time (avg s)	1.040703	1.152488	1.25908	1.3222558	1.737544

Table 2: Kernel-space results.

4.2 Physical Machine

The following tables show the result of the execution on the physical machine.

	#Fibers	16	64	128	512	1024
Monoprocess	Init time (s)	0.0010000	0.0010000	0.0040000	0.0150000	0.0120000
	Running time (s)	0.3418130	0.3289530	0.3288360	0.3566000	0.3526230
Multiprocess (4)	Init time (avg s)	0.0140000	0.0010000	0.0242500	0.0280000	0.0217500
	Running time (avg s)	1.1720468	1.2499963	1.2693193	1.5069943	1.7538790

Table 3: User-space results.

	#Fibers	16	64	128	512	1024
Monoprocess	Init time (s)	0.0000000	0.0010000	0.0010000	0.0050000	0.0070000
	Running time (s)	0.3694370	0.3689690	0.3648910	0.3689900	0.4343290
Multiprocess (4)	Init time (avg s)	0.0000000	0.0347500	0.0095000	0.0315000	0.0260000
	Running time (avg s)	1.4388908	1.4374805	1.6085525	1.6440565	2.1317930

Table 4: Kernel-space results.

The results show that, overall, the user-space implementation is slightly faster in running the Fibers while the kernel-space one shows slightly better performances in the Fiber initialization.