Operating systems

Academic year 2019-2020

Project 4
Drawing fork

Bolland Julien - Gilson Maxence

(s161622 - s162425)

Inside fork() implementation

In this section, we will refer to the diagram provided with this report, named `Diagram.pdf`. All of the functions discussed below have been retrieved from Linux v4.15.0 kernel files and their exact location can be found in the appendice.

The diagram is based on two part, the user space (in green) and the kernel space (in red). Let's assume that a user has created a program in C that calls the function fork. This function will be relayed to a specific C library, called a wrapper, that makes a correspondence between the function called by the user and the systems call available. Then, the wrapper will execute a command that will make a switch to kernel mode. In this mode, the kernel checks the entry point of the called function and executes its code.

1 _do_fork

This function is the main implementation of fork. For schematic reasons, its arguments have not been represented in the diagram. Therefore, they are represented in the following table, along with there value when fork is called.

Name	Use	Initial value
clone_flags	Flags used in order to know the cloning options	SIGCHLD
stack_start	The starting point of the stack	0
stack_size	The size of the stack	0
parent_tidptr	Pointer on the parent caller	NULL
child_tidptr	Pointer on the child created	NULL
tls	Thread-local stack	0

TABLE 1 – Arguments of _do_fork

It is not useful to describe all of its arguments. Nevertheless, it can be noticed that the starting point of the stack is set to 0, since each process "sees" itself as having access to all the computer's memory. The stack size is set to 0 because the child's stack has not been initialised yet.

The function begins by initialising a pointer on a `task_struct`, called p, which is a structure containing useful information for a process in Linux. Then, the function assigns to p the return value of a function, named `copy_process` (described in section 1.1).

After this call, the function wakes up the newly created process with `wake_up_new_task` (described in section 1.3). Then, it returns the value of the PID of the new process to the parent, so that it can identify its child easily.

1.1 copy_process

This function is used to copy the registers and the states of the parent process inside a new process structure. This is where the kernel copies the page files of

the parent and assigns them to the child (cf. below for further explanation). To do so, it uses a another function, called `dup_task_struct` (described in section 1.2), to duplicate the corresponding parent process.

Then, the function assigns to the new process structure every data that are related to a new process (*i.e.* scheduler data, information for the CPU,..). A assignment to notice, however, is the one written on the diagram : `p->pagefault_disable = 0`. Indeed, this shows us that the kernel enables to the new process a page fault mechanism. This will be used in the CoW concept. Moreover, the function enables the process to be known by the computer but it is important to notice that it is not the role of this function to wake up the process.

After all the assignments to the new process, the function returns this new process structure to the calling function.

1.2 dup_task_struct

This function allows the kernel to allocate memory for a process structure, *i.e.* `task_struct`, and for a stack. This stack will be the stack used by the newly created process. We can see on the diagram two functions with the names beginning by `alloc`, but we won't describe them here, since their behavior is clear.

1.3 wake_up_new_task

We won't go too much on the details of this function. Its aim is to perform some initial scheduler statistics. Then, it puts the task on the run queue and wakes it.

Overall point of view

When the function fork is called by a process (the parent), the operating system creates a new process (the child). At the end of the function, if we assume it to be successful, the child shares the same page files as its parent. The return value of fork is different for the child and the parent. The parent gets the PID of the child newly created, whereas the child is returned 0.

The page sharing between the parent and the child is an implementation choice. In fact, this reduces the time needed for the computer to execute the function, because it won't be necessary to copy all the content of the parent's frames to new frames.

This behavior is showed on the first and second box of Figure 1. [1] On the first box, we can see a process named P1, the parent process. This process has a PID and a variable a, set to 1. In order to be schematic, P1 only has 2 frames (A and B) in its paging file. Let's assume that frame A is responsible for the process

1. For the rest of this report, we will denote as the first box the black rectangle located at the top left corner of Figure 1, the second box as the one located at the top right corner, the third as the one located below the first box and the fourth as the one located below the second box.

metadata (stack, PID and every other process related data) and that frame B is responsible for the variable a. The arrow [1] represents the action of P1 calling the function fork. After this function call, a new child process P2 is created. As explained above, this new process has two frames, which are the same as P1. However, during the fork, these two frames are set to read only. Indeed, the aim of fork is to perform multi-threading, which means that parent and child have to work concurrently. If one modifies a for instance, the other will handle the modified value of a, because both processes would share the same frames of the same paging file. This could lead to inconsistencies in some applications.

Copy-on-Write

Because of the multi-threading issue discussed above, fork function sets the frames used by both processes to read only. However, the processes can still perform operations. This is done thanks to the Copy-on-Write (CoW) concept. Whenever a process wants to perform an operation, the kernel will detect the read only frame that the process wants to write on. Then, it will create an exact copy of this frame in the paging file, linking to another location in the physical memory.

CoW behavior is shown in boxes 3 and 4. Let's assume, for instance, that process 2 wants to assign the value 100 to the variable a (done with action [2]). The kernel will detect the read only property of frame B. Then, the kernel trap handler will copy frame B into a newly created frame B'. This new frame has its permission changed to read-write (action [3]). After this operation, the kernel returns the program counter to the previous statement to try the assignment of $a = 100$ once again. From now on, P2 and P1 don't share the same frame containing a link to the value of a in physical memory. Therefore, P1 "sees" a as being equal to 1 and P2 "sees" $a = 100$.

Appendice

2 Figures

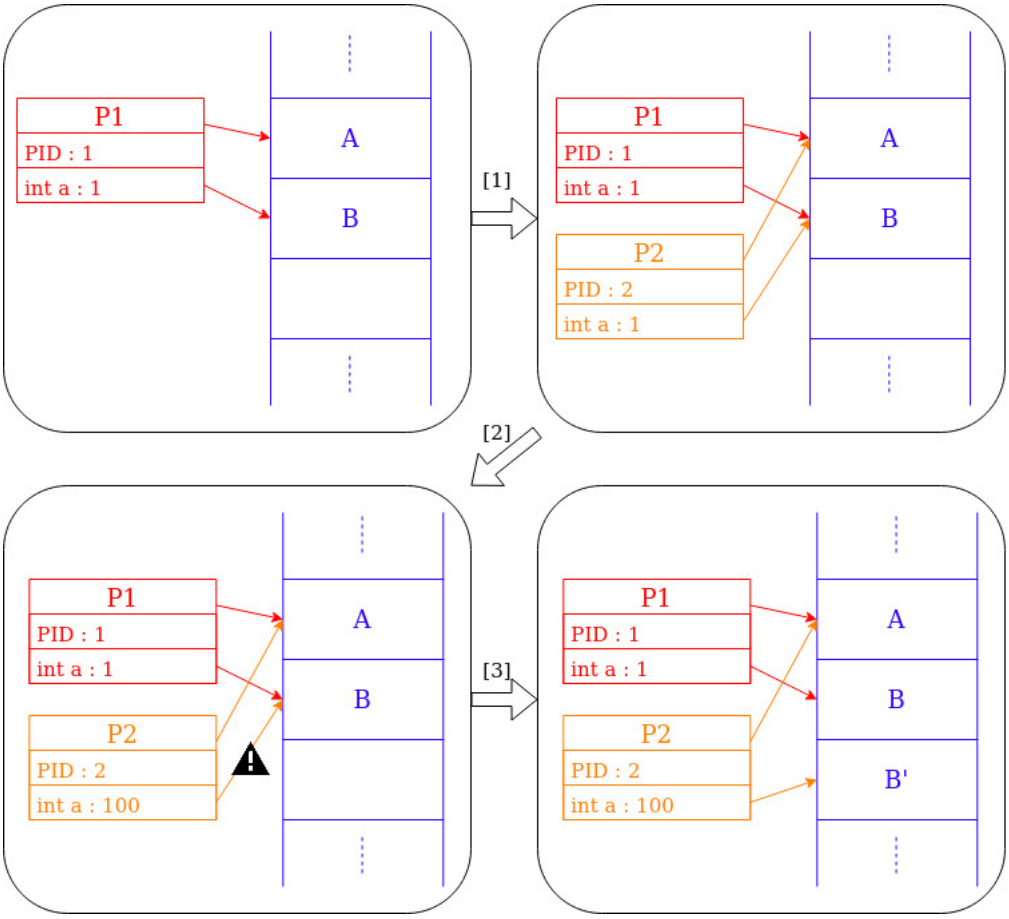

FIGURE 1 – Representation of a fork call and the Copy-on-Write behaviour

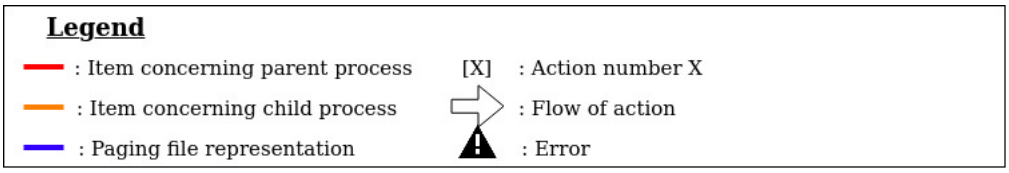

FIGURE 2 – Legend of Figure 1

3 Location of functions

All of the functions below have been checked in the kernel provided by the teaching assistant of this course (Linux v4.15.0). The provided lines are the ones in which the function is implemented.

Name	Location	Lines
`task_struct`	/include/linux/sched.h	520 to 1106
`_do_fork`	/kernel/fork.c	2015 to 2085
`copy_process`	/kernel/fork.c	1534 to 1984
`dup_task_struct`	/kernel/fork.c	512 to 595
`wake_up_new_task`	/kernel/sched/core.c	2447 to 2485
`alloc_task_struct_node`	/kernel/fork.c	154 to 157
`alloc_thread_struct_node`	/kernel/fork.c	204 to 246 OR 272 to 276 ([2])

TABLE 2 – Location of each functions and structure used in this report

2. Depending on CONFIG_VMAP_STACK