University of Liège

Project 4 : Drawing the fork primitive

Operating systems

Maxime Meurisse (s161278)
Valentin Vermeylen (s162864)

Master in Engineering and Computer Science
Academic year 2019-2020

1 Introduction

In the context of the version of Linux installed on the course reference virtual machine (Linux 4.15)[1], *fork* is an operation that allows a process to create a copy of itself. This operation is usually used in combination with the system call *exec* so that the newly created process, called *child process*, can execute another operation.

An important point of the fork operation is *memory management*. This document highlights first the different major operations taking place in the Linux kernel when calling a fork and then how the memory, and more specifically that of the child process, is managed.

The various explanations are supported by a diagram presented in the appendix of this document and also attached as a `diagram.pdf` file.

2 Implementation

The fork system call is not implemented in *sys_ call* any longer. It actually calls the *clone* function (a system call to create a new thread) which itself calls the *_ do_fork* function that does the work of creating a new process. The latter is the actual implementation of the fork operation. It calls the function *copy_process* which performs all the main operations to create the new process.

2.1 Main operations

Main operations performed by the function *copy_process* are listed below.

1. *The copy_process function* : this function creates a new process as a copy of the parent, but does not start it. It copies the registers and the appropriate parts of the process environment, but the starting of the new process is left to the caller.

   The function first checks that the flags passed as parameters are compatible, then calls the function *dup_ task_ struct*.

2. *The dup_task_ struct function* : this function is used to create new kernel stacks for the new thread that is being created, as well as allocate the *task_ struct* used to represent the new process (this structure is used to store a lot of state information), and the *thread_ info* structure. All those values are the same as the ones of the parent process (the origin of the call to fork).

   After that, process descriptor are cleared or set to initial values. It also sets the usage counter of the new *task_ struct* to 2 so that the operating system knows the corresponding process is alive and not dead or in a zombie or orphan state.

   A function *copy_flags* is called to update the flags member of the *task_ struct* structure. Depending on the flags, what resources are to be shared is decided.

   Finally, that data structure is returned at the end of the function.

3. *End of copy_process* : once the task structure has been allocated and correctly filled, a return stack is allocated to the newly created task. The function then goes

[1]All explanations in this document will refer to this version of Linux.

to set all fields of the newly created structure appropriately and returns to the caller a pointer to the newly created child.

4. *Coming back to _do_fork* : finally, after checks to see if the new thread pointer is not invalid, the process is awoken.

It is in step 2 that the main elements of the child process are allocated and copied from the parent process. The different memory management mechanisms are presented in the next section.

3 Memory management

3.1 Overview

When the fork operation is called, it creates a new address space for the child process. In theory, the two processes therefore work in two distinct address spaces and therefore whatever operation is performed by one process will not affect the other process.

As mentioned in step 2 of the implementation, it is the *task_struct* structure that is copied. This structure is called a *Process Control Block* (PCB). It contains a lot of state information : pid, user id, register state, process status, etc. Not all fields in the structure are strictly copied; some, such as pid or process status, are obviously not copied from the parent. All the memory segments of the parent process is also duplicated. These two elements (memory and PCB) characterize a process in the kernel.

In practice, this would mean copying a lot of memory elements each time an exec is called by a child process created with fork. To avoid this, Linux uses a technique called *Copy-on-write* (COW) : the parent and child processes share a single copy of the process address space and only when one of them tries to write into the process address space, a new copy is created for that process.

This technique avoids copying large segments of memory each time an exec is called by a child process.

3.2 Copy-on-write

Copy-on-write is a technique to efficiently copy data resources in a computer system. The principle is as follows : when a resource is duplicated but not modified, it is not necessary to create a copy. As the name of the technique suggests, a copy is created only when a write to memory is performed.

It can be implemented efficiently by using the page table : the pages of memory of the parent process are marked as read-only and the kernel keeps a count of the number of references to the page.

If both processes only read, they work with shared memory, but when a process (parent or child) tries to write to one of these pages, it creates a *page fault* which is intercepted by the kernel trap handler. The latter then creates a new physical page (except if there is only one reference to this page), changes the permission in *read-write*, updates the page table, decrements the number of references and returns the *Program Counter* (PC) to the previous statement. The latter is then re-run and the writing in memory (on the newly created page) can be done.

Thanks to this new allocated page, a change made in one process will not be visible in another.

3.3 Cases of failure

The fork operation can fail in some situations. Concerning memory, the operation may fail if no more memory is available (*out of memory*) or if the parent process consumes more than 50% of the system memory.

In the latter case, the creation of the child process would not be especially problematic (since not all the memory of the parent process is directly copied) but the interaction with the elements of the memory by the child process would be problematic (the elements would then be copied by the COW mechanism but would in fact not have enough memory to be copied).

3.4 File descriptor

As mentioned in step 2 of the implementation, process descriptors are cleared and set to initial values. However, the file descriptor of the parent is not duplicated when the child process is created. This implies that resources can be shared between the different processes. Some anonymous objects (pipes, shared memory, etc.) can only be shared through this mechanism.

Diagram

The diagram is shown in Figure 1.

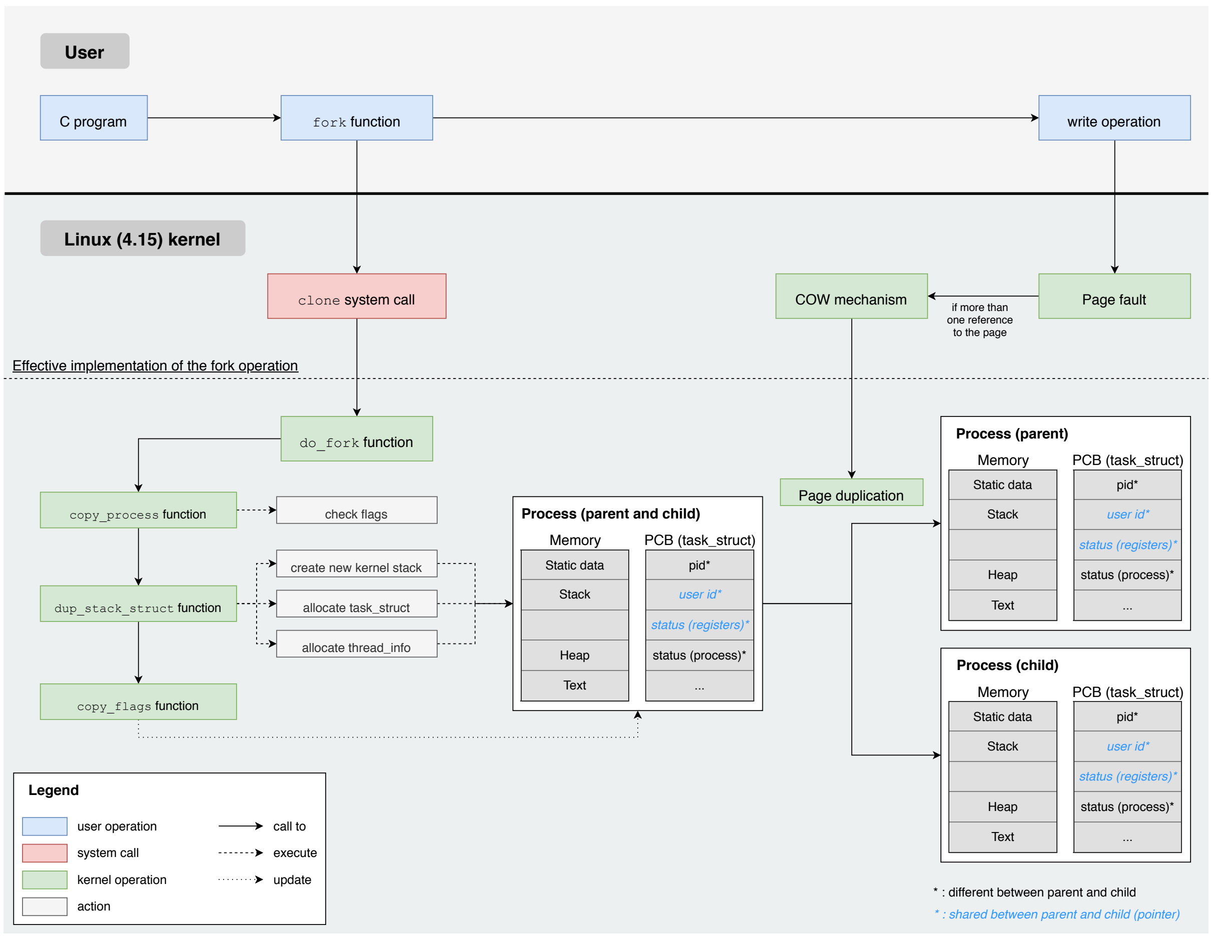

Figure 1: Diagram of the fork operation in the Linux kernel.

References

[1] Wikipedia. *Fork (system call)*. URL: `https://en.wikipedia.org/wiki/Fork_(system_call)`. (accessed : 30.04.2020).

[2] Pavan Koli. *The fork() system call*. URL: `https://www.hackerearth.com/fr/practice/notes/the-fork-system-call/`. (accessed : 30.04.2020).

[3] Wikipedia. *Copy-on-write*. URL: `https://en.wikipedia.org/wiki/Copy-on-write`. (accessed : 30.04.2020).

[4] Prof. Bershad. *Processes, Loaders, Fork, and Exec*. URL: `https://courses.cs.washington.edu/courses/cse451/02sp/section/notes/fork/`. (accessed : 30.04.2020).

[5] Ayesha Tariq. *Fork System Call Linux*. URL: `https://linuxhint.com/fork-system-call-linux/`. (accessed : 30.04.2020).

[6] Liran B. H. *Linux – Fork system call and its pitfalls*. URL: `https://devarea.com/linux-fork-system-call-and-its-pitfalls/`. (accessed : 30.04.2020).