VForkExecutor

*Programming in UNIX systems 202*0
Michal Stefaniuk

Project target

The goal of the project was to create a program that would execute given compiled another program using vfork() function and compare it to the usual execution in the shell or with the fork.

For me the most difficult thing was to actually understand the workflow of system callers in UNIX systems and the whole workflow. When I got familiar with the idea it was actually fun to create this project.

How to run

Program has attachted `Makefile` that is responsible for building the project. Clone the repository to your local device and follow below instructions.

Build the project type

```
make
```

Clean the project

```
make clean
```

Run the project

```
make run
```

When you run the project a mockup shell will appear

```
VForkExecutor $:
```

Type any program's name to execute it as a child process created by `vfork()`. My proposition for the project is to use bubble-sort algorithm stored in **sort.c**, for example

```
VForkExecutor $: ./sort type_of_system_call
```

There are three types of system calls for user to choose from. * fork

> Printing array using process created by `fork()` should present correct and simultaneous execution. **while (wait(&status) != pid)** is used to wait for the parent to complete * vfork Printing array using process created by `vfork()` should present that the parent process will be suspended until child execution is completed, although **exit(0)** needs to be called as child has to finish with exit

- vfork_err >Printing array using process created by `vfork()` but this time no **exit(0)** is called so the behaviour is not predictable and the most likely result would be **segmentation fault**

To exit the program just type

```
VForkExecutor $: exit
```

Workflow description

Each one of system calls implements different behaviour. Example usage

```
VFrokExecutor $: ./sort fork
```

This will run `./sort` in child process created in `./fork`. Then user is asked to input an array. Next task will be bubble sorting given array and regarding type of system call, array will be printed in another process created by **fork()** or **vfork()**

To use `vfork()`

```
 VFrokExecutor $: ./sort vfork
```

And to see how unpredictable `vfork()` can be

```
 VFrokExecutor $: ./sort vfork_err
```

Implementation chart

Performance of system calls

- `fork()` - creates a new child process, which is a complete copy of the parent process. Child and parent processes use different virtual address spaces, which is initially populated by the same memory pages. Then, as both processes are executed, the virtual address spaces begin to differ more and more, because the operating system performs a lazy copying of memory pages that are being written by either of these two processes and assigns an independent copies of the modified pages of memory for each process. This technique is called Copy-On-Write (COW).
- `vfork()` - creates a new child process, which is a "quick" copy of the parent process. In contrast to the system call `fork()`, child and parent processes share the same virtual address space. NOTE! Using the same virtual address space, both the parent and child use the same stack, the stack pointer and the instruction pointer, as in the case of the classic `fork()`! To prevent unwanted interference between parent and child, which use the same stack, execution of the parent process is frozen until the child will call either `exec()` (create a new virtual address space and a transition to a different stack) or `_exit()` (termination of the process execution). `vfork()` is the optimization of `fork()` for "fork-and-exec" model. It can be performed 4-5 times faster than the `fork()`, because unlike the `fork()` (even with COW kept in the mind), implementation of `vfork()` system call does not include the creation of a new address space (the allocation and setting up of new page directories).
- `clone()` - creates a new child process. Various parameters of this system call, specify which parts of the parent process must be copied into the child process and which parts will be shared between them. As a result, this system call can be used to create all kinds of execution entities, starting from threads and finishing by completely independent processes. In fact, `clone()` system call is the base which is used for the implementation of `pthread_create()` and all the family of the `fork()` system calls.
- `exec()` - resets all the memory of the process, loads and parses specified executable binary, sets up new stack and passes control to the entry point of the loaded executable. This system call never return control to the caller and serves for loading of a new program to the already existing process. This system call with `fork()` system call together form a classical UNIX process management model called "fork-and-exec"

According to above `vfork()` function has the same effect as `fork()`, except that the behavior is undefined if the process created by `vfork()` either modifies any data other than a variable of type `pid_t` used to store the return value from `vfork()`, or returns from the function in which `vfork()` was called, or calls any other function before successfully calling `_exit()` or one of the `exec()` family of functions.

So by calling `printf()` in your child code, your code is already undefined.

> Note that even a call to `exit()` could sometimes lead to undefined behavior

Qutoe from **LINUX man**

> `vfork()` differs from `fork(2)` in that the calling thread is suspended until the child terminates (either normally, by calling `_exit(2)`, or abnormally, after delivery of a fatal signal), or it makes a call to `execve(2)`. Until that point, the child shares all memory with its parent, including the stack.

`fork()` VS `vfork()`

	frok()	vfork()
Address space	Both the cild and parent process will have different address space	Both child and parent process share the same address space
Modification in address space	Any modification done by the child in its address space is not visible to parent process as both will have separate copies	Any modification by child process is visible to both parent and child as both will have same copies
CoW(copy on write)	This uses copy-on-write	Doesn't use CoW
Execution summary	Both parent and child executes simultaneously	Parent process will be suspended until child execution is completed
Outcome of usage	Behaviour is predictable	Behaviour is not predictable

Benchmark

The project has built in `vfork()` and `fork()` time benchmarking mechanism. The workflow goes as below: * **fork.c** creates in loop child process *50 times* * **benchmark.c** generates *5-element* array filled with random values <0,50> * time of processing the array is counted and used for further steps

First we want to output times of execution to txt files

```
make run > benchmark_vfork.txt
./benchmark vfork
exit
```

Do the same with `fork()`

```
make run > benchmark_fork.txt
./benchmark fork
exit
```

This will result in two .txt files that we want to merge into one file. To do it we will use python script

```
python parser.py
```

And now we have **benchmark.txt** file which is ready to be plotted. To do so we will use gnuplot script which will present results on image graph

```
gnuplot plot.sh
```

And the results are here

As we see `vfork()` performs way faster. But at this point we are not surprised, are we?