Report on exercise #1

Matteo Corain S256654

Laboratory #6 – System and device programming – A.Y. 2018-19

In order to list and comment the system calls performed when the proposed command has been issued, it is first necessary to understand how the system calls mechanism is handled in the xv6 kernel. In particular, after a quick inspection of the code, it is possible to notice that all the system calls are run by means of the `syscall()` function, defined in `syscall.c`.

This function receives in the EAX register of the processor the number of the system call that has been requested, and then runs the respective `sys_` system function by accessing a static array of function pointers to the different system functions, called `syscalls`, storing the result again in the EAX register; the indices of all the different system calls are defined in `syscall.h`. This function is called in turn by the `trap()` function, defined in `trap.c`, when a system call-related trap is generated.

The `syscall()` function has been exploited in order to list the sequence of system calls originated by issuing the proposed command; in particular, a breakpoint has been placed inside the function in order to take note of the number of the requested system call. This task has been automated by placing, at the end of the `.gdbinit` file, the following lines:

```
break syscall.c:145
commands 1
print num
continue
end
continue
```

The effect of those lines is to print the value of the variable num whenever the breakpoint at line 145 of `syscall.c` is reached, then continue the execution of the program. The output of `gdb` has then been redirected to a log file, from which the sequence of system calls numbers has been extracted by means of a pipe of `grep` and `cut` operations.

In order to understand the behavior and the purpose of each system call, it has been necessary to inspect the code of the functions executed by the `sh`, `cat` and `wc` programs when a command is issued. In particular, the following actions are taken:

- The shell reads the user input by means of the `getcmd()` function, used as the condition for an infinite `while` loop;
- The shell parses the user command by means of the `parsecmd()` function, which detects the type of command (`EXEC`, `REDIR`, `PIPE`, `LIST`, `BACK`) to be executed and allocates a proper structure to describe it;
- The shell executes the parsed command in a forked process by means of the `runcmd()` function, which behaves differently based on the type of command, and then waits for it to complete; in our case, the command is a `PIPE`-type command, and therefore the relative `case` of the `switch` construct is executed by the forked process;
- The forked shell opens a pipe for making the processes running the two sides of the pipe communicate with each other;
- The forked shell forks two child processes, in which:

    - In the first one, the `stdout` file descriptor (`1`) is closed, and in its place the write terminal of the pipe (`p[1]`) is dup-ed; after that, the file descriptors of the pipe are closed (the write terminal has been duplicated on file descriptor `1`, so the reference count for that descriptor is still positive) and a recursive `runcmd()` call is performed to run the left part of the piped command (the command before the pipe);
    - In the second one, the `stdin` file descriptor (`0`) is closed, and in its place the read terminal of the pipe (`p[0]`) is dup-ed; after that, the file descriptors of the pipe are closed (the read terminal has been duplicated on file descriptor `0`, so the reference count for that descriptor is still positive) and a recursive `runcmd()` call is performed to run the right part of the piped command (the command after the pipe).
- The forked shell closes the two pipe descriptors and waits for the completion of the two children; in concurrency, the two children execute the `runcmd()` function (in this case, the command is an EXEC-type command), which performs an `exec()` operation to change their code into the one of the requested program (without changing the file descriptors, those are inherited from the parent);
- The `cat` program, executed in the first child process, performs the following operations:
    - It opens the file given as an argument on the command line;
    - While the `read()` operation on the given file returns a value higher than `0` (no more data is present in the file), it writes the content of the read buffer to its standard output (which, in this case, has been replaced with the write terminal of the pipe);
    - It closes the input file.
- The `wc` program, executed in the second child process, performs the following operations:
    - Since in this case no file is given, the input is assumed to be the standard input (which, in this case, has been replaced with the read terminal of the pipe);
    - While the `read()` operation on `stdin` returns a value higher than `0` (no more data is present), the program performs the counting of characters, words and lines.
- After the two children has completed, the forked shell process terminates as well; the shell then restarts the execution from the `getcmd()` operation.

It is now possible to present the list of the performed system calls, in their execution order (it may change due to the scheduling of processes running in concurrency).

Number	Name	Program	Description	References
5	`read`	`sh (#1)`	The shell reads the user input, one character at a time (executed 20 times in sequence), via the `gets()` function.	`sh.c`: • `main()` • `getcmd()` `ulib.c`: • `gets()`
1	`fork`	`sh (#1)`	The shell forks a process to execute the parsed user command.	`sh.c`: • `main()`
3	`wait`	`sh (#1)`	The shell waits for the completion of the child process, which executes the user command.	`sh.c`: • `main()`
12	`sbrk`	`sh (#2)`	The process memory for the child is grown by an additional 32 kB (it was 16 kB).	None
4	`pipe`	`sh (#2)`	The forked shell creates a pipe for executing the command.	`sh.c`: • `runcmd() – case PIPE`

1	fork	sh (#2)	The forked shell forks the child to execute the left-side program.	sh.c: • runcmd() – case PIPE
1	fork	sh (#2)	The forked shell forks the child to execute the right-side program.	sh.c: • runcmd() – case PIPE
21	close	sh (#3)	The first child closes the stdout descriptor.	sh.c: • runcmd() – case PIPE
10	dup	sh (#3)	The first child duplicates the pipe's write terminal to descriptor 1.	sh.c: • runcmd() – case PIPE
21	close	sh (#3)	The first child closes the descriptor of the read terminal of the pipe.	sh.c: • runcmd() – case PIPE
21	close	sh (#3)	The first child closes the descriptor of the write terminal of the pipe.	sh.c: • runcmd() – case PIPE
7	exec	sh (#3)	The first child executes the cat program.	sh.c: • runcmd() – case EXEC
21	close	sh (#2)	The forked shell closes the descriptor of the read terminal of the pipe.	sh.c: • runcmd() – case PIPE
21	close	sh (#2)	The forked shell closes the descriptor of the write terminal of the pipe.	sh.c: • runcmd() – case PIPE
3	wait	sh (#2)	The forked shell waits for the first child to complete.	sh.c: • runcmd() – case PIPE
21	close	sh (#4)	The second child closes the stdin descriptor.	sh.c: • runcmd() – case PIPE
10	dup	sh (#4)	The second child duplicates the pipe's read terminal to descriptor 0.	sh.c: • runcmd() – case PIPE
21	close	sh (#4)	The second child closes the descriptor of the read terminal of the pipe.	sh.c: • runcmd() – case PIPE
21	close	sh (#4)	The second child closes the descriptor of the write terminal of the pipe.	sh.c: • runcmd() – case PIPE
7	exec	sh (#4)	The second child executes the wc program.	sh.c: • runcmd() – case EXEC
15	open	cat (#3)	The input file is opened.	cat.c: • main()
5	read	cat (#3)	The input file is read to an internal buffer.	cat.c: • cat()
16	write	cat (#3)	The contents of the internal buffer are written to stdout.	cat.c: • cat()

5	read	cat (#3)	The input file is read again (this time, no data is present).	cat.c: • cat()
21	close	cat (#3)	The input file is closed.	cat.c: • main()
2	exit	cat (#3)	The program terminates the execution.	cat.c: • main()
3	wait	sh (#2)	The forked shell waits for the second child to complete.	sh.c: • runcmd() – case PIPE
5	read	wc (#4)	The standard input is read to an internal buffer.	wc.c: • wc()
5	read	wc (#4)	The standard input is read again (this time, no data is present).	wc.c: • wc()
16	write	wc (#4)	The results are written to stdout via the printf() function, one character at a time (executed 7 times).	wc.c: • wc() printf.c: • printf()
2	exit	wc (#4)	The program terminates the execution.	wc.c: • main()
2	exit	sh (#2)	The forked shell terminates.	sh.c: • runcmd()
16	write	sh (#1)	The shell prints "$ " on the screen via printf() (executed twice, once per character).	sh.c: • runcmd() printf.c: • printf()
5	read	sh (#1)	The shell waits for the user to input the next command.	sh.c: • getcmd()