Operating Systems
Jacobs University Bremen
Dr. Jürgen Schönwälder

Course: CO20-320202
Date: 2018-09-10
Due: 2018-09-27

OS 2018 Problem Sheet #1

Problem 1.1: *simple cat (scat) using library and system calls* (5 points)

Write a program `scat` (simple cat or slow cat) that copies data from the standard input to the standard output.

a) Implement the data copying loop using the C library functions `getc()`/`putc()` and using the system calls `read()`/`write()` operating on a single byte at a time. Your `scat` program should accept the command line options `-l` and `-s`: The option `-l` selects the C library copy loop (using `getc()`/`putc()`) and the option `-s` selects the system call copy loop, using one byte at a time `read()`/`write()` system calls. In case there are multiple options on the command line, the last option wins. If there is neither a `-l` nor a `-s` option, the program uses the C library functions copy loop.

b) Use your `scat` program to copy a large file to `/dev/null` (a device file that discards all data) and measure the execution times:

```
time ./scat -l < some-large-file > /dev/null
time ./scat -s < some-large-file > /dev/null
```

Repeat the measurements a few times to get stable results. What do you observe? Explain. Use strace to investigate the read/write sizes that are used by the two variants of your program. How many read/write calls in total are executed while copying your large file?

c) Extend your program by implementing another option `-p` and a copy loop that uses the Linux specific `sendfile()` system call. Set the amount of data that is copied in each call of `sendfile()` such that it matches the number of bytes read and written by the C library. Measure the execution time.

```
time ./scat -p < some-large-file > /dev/null
```

What do you observe? Explain.

Hand in the source code of your `scat` program and the results of your analysis. Make sure that your program handles *all* error situations appropriately. Use the `getopt()` function of the C library for the command line option parsing.

Problem 1.2: *watch - execute a program periodically* (5 points)

Write a C program called `watch` that executes a command periodically (e.g., every 2 seconds), showing the output on the standard output (terminal). Your implementation of `watch` does not have to clear the screen like other implementations of `watch` do. Your program should implement a command line option `-n` to set the number of seconds that `watch` sleeps between each repeated execution of the command. The option `-b` causes the special value
`a` to be written to the standard output if an execution of the command ends with a non-zero exit code (this usually rings the terminal bell). The option `-e` terminates your `watch` program when the execution of a command fails. (If `-e` is not given on the command line, the execution continues irrespective of any failures of the command execution.)

Your program must use the `fork()`, `execvp()`, and `waitpid()` system calls. You are not allowed to use the `system()` library call. You can let your `watch` program sleep by calling the `sleep()` library function.

```
$ ./watch date
Tue Sep 13 13:51:33 CEST 2016
Tue Sep 13 13:51:35 CEST 2016
Tue Sep 13 13:51:37 CEST 2016
Tue Sep 13 13:51:39 CEST 2016
```

```
$ ./watch -e ls -l /foo
ls: /foo: No such file or directory
$
```

Make sure your program properly handles all possible runtime errors and that it returns an error status to its parent process (usually the shell) in case a runtime error occured.

Use the `getopt()` function of the C library for the command line option parsing.