Lab 3: Processes

Introduction to Studies in Embedded Systems (1DT086)

—

Advanced Computer Science Studies in Sweden (1DT032)

—

Autumn 2019, Uppsala University

1 Introduction

1.1 Goals

In this lab you will practice some basic process management: creating processes and executing other programs from C on Linux. You will also practice controlling the LED matrix of the Sense HAT from C.

1.2 Preparations

Read through these instructions **carefully** in order to understand what is expected of you. You should have finished *Lab 2* and have a working Linux system on your Raspberry Pi before doing this lab.

1.3 Report

For this lab you have to hand in a written report. Each group hands in one joint report, where *all* group members have participated in *all* parts of the report, and can individually explain every part if asked. These instructions will make it clear what should go into the report.

The report *must* be handed in in PDF format, and be written in LaTeX using the template `report-template.tex` that is found on the Student Portal.

2 Warming up

This warming-up section does not contain anything that will go into the report, but it is highly recommended that you go through it.

2.1 Some process management

User-space programs (e.g., text editor, web browser or ssh daemon) execute in one or more *processes*. Processes are mostly isolated from one another: they have their own address space, environment variables, open file descriptors etc. Each process also contain one of more *threads*, which are actually running the program code. Simple programs are typically executed in a single process, but some programs spawn several processes when run (an example of this is `sshd`—more on this later).

Log in to your Raspberry Pi using SSH and use the `pstree` program to list the processes currently running on it.

```
pstree
```

This program represents the processes as a tree, where processes further down the branches have been spawned by the process preceding it. The root of the tree is a process called `systemd`, which on this system was the first user-space process that was created.[1] The program `pstree` is itself represented by a process in the tree, created by the process `bash`, which is the default shell that was started when you logged in. In turn, the process `bash` was created by a process `sshd`, which was also created by a process named `sshd` and so on.[2] You can also note a process named `wpa_supplicant` handling Wifi connections and a process `cron` for the Cron daemon running scheduled tasks, like updating your freemyip.com domain name.

Clearly, process names are not unique as there are several processes named `sshd`. Every process instead has a unique number, its Process ID (or PID). Run `pstree` again with the `-p` flag to have it print also the PIDs within parenthesis after the process names.

```
pstree -p
```

Without logging out of your SSH session, start another SSH session and

[1] Traditionally this process is called `init`, but not so on systems running systemd.

[2] This is just how `sshd` works, the top process named `sshd` is the one listening to new connection, the next one is handling an established connection, sending commands to the third one for execution. These processes have different privilege levels, so this approach offers some security benefits, in addition to some modularity.

log in. Run `pstree -p` and note how the topmost `sshd` process has spawned another branch for the new connection.

Note the PID of the other `bash` process in the output. Terminate it using the `kill` command, replacing `<PID>` with that processes PID.

```
kill <PID>
```

Note how `bash` prints "`logout`" before terminating, at which point SSH client prints "`Connection to N.N.N.N closed.`" The `kill` program without additional flags sends a signal to the process that it should terminate, which `bash` in this case does cleanly (printing "`logout`" while doing so).

Log in with a second SSH session again, and repeat the above to terminate the `bash` process, but now issuing the `kill` command with the `-9` flag.

```
kill -9 <PID>
```

Note how this again closes the SSH connection, but `bash` does not print "`logout`". This is because `kill -9` tells the kernel to immediately terminate the process, which does not give the process a chance to exit cleanly (but on the other hand can terminate it without the process' cooperation).

Last, play around a bit with the `htop` command and try to, for example, terminate some process from its interactive interface.

2.2 Using the LED matrix

The LED matrix of the Sense HAT add-on board appears in Linux as a *framebuffer device*. The framebuffer device is a special file that you can write to, much like any other file. However, writing to this special file does not result in the values being stored on a storage device, but instead it changes the colors of the LEDs.

On the Linux systems on your Raspberry Pi's, this framebuffer device should[3] be mapped to the (special) file `/dev/fb1`. The directory `/dev/` usually contain only special device files of different kinds.

Try writing some data to the `/dev/fb1` file. For example you can write any string of characters to it using the `echo` command, and then you should be able to see the colors of the LEDs change.

[3]For reasons unbeknownst, it is sometimes given another name. If you don't have the framebuffer device file `/dev/fb1`, try to find which other filename it was given and replace `/dev/fb1` with that from here on.

```
echo "Hello" > /dev/fb1
```

Warning: Don't accidentally write to other files in `/dev/`. Doing so could cause issues like corrupting your microSD card!

If you set up everything correctly in the last lab, your ordinary users should have permissions to write to this file. If not, go back to those instructions and make sure that you get those permissions. In particular, your users have to belong to the appropriate groups.

Writing to `/dev/fb1` should change the colors of the LEDs. The colors that you get depends on what you write to the file. The LED matrix works by having 16 bits per LED that encode the color in RGB565 format. The RGB565 format encodes a color into 16 bits by having the 5 most significant bits encode the *red* channel, the next 6 bits the *green* channel and the last 5 bits the *blue* channel.

When you write a string of characters to the framebuffer file file using `echo`, the sequence of bits representing those characters will simply be interpreted as encoding colors in the RGB565 format. Of course, writing strings to the file is a poor way of controlling what is displayed on the LED matrix. Fortunately, you can instead open the file from within a program that you write and control the LEDs in this way.

The most convenient way of doing this is to open the framebuffer file in your program and then *map* its contents into your processes' memory address space using the `mmap` system call. Doing so will cause a region of the address space to contain the contents of the file. Writing data to that memory region then writes the data to the file, which will in turn change the colors of the LEDs.

If you wish, you are free to do this mapping of the LED matrix framebuffer file by yourselves in what follows of this lab. However, there is a small C library that you can use that does the mapping/unmapping for you, as well as providing a number of convenience functions for controlling the LED matrix and for creating your own RGB565-encoded colors.

Download the file `led_matrix_example.tar.gz` from the Student Portal, and copy it to your Raspberry Pi using `scp`. On the Raspberry Pi, unpack it using the `tar` program.

```
tar -xvf led_matrix_example.tar.gz
```

In the unpacked directory are the files `led_matrix.c` and `led_matrix.h`. These make up the tiny library that you can use for controlling the LED matrix from within your C programs. There is also a demo program `led_example.c`, which uses the library to display some things on the LED matrix. Study these three files to understand what they do, in particular the demo program `led_example.c` and the header file `led_matrix.h`. The latter contains defini-

tions of all the functions from `led_matrix.c` that you can use, as well as some predefined RGB565 colors.

There is a `Makefile` for compiling the `led_example` program. Use it to compile by issuing the `make` command from within the directory. Then run the `led_example` program to see that it works as expected.

```
make
./led_example
```

In the following, you are free to use `led_matrix.c` and `led_matrix.h`. You are also free to take inspiration from, or reuse parts of, `led_example.c` and its `Makefile` as you see fit.

A convenient property of memory-mapped files like used in the `led_matrix` library is that the mapping is preserved in the new process after a call to `fork()`. This makes memory-mapped files a possible method for data sharing between processes. For this particular case, it also allows several processes in your programs to access the LED matrix if the parent has set up the mapping before creating the children. This property you will use later in this lab.

3 *Task 1:* Creating some processes

It is now time to write programs that create some processes. First, answer the following the questions in your report.

Question 1.1 —— *How many new processes will a program that contains the code* `fork(); fork(); fork(); fork();` *create? Explain your answer clearly.*

Question 1.2 —— *What is the difference between the code snippets in the below listings with respect to the parent-child relationship of the created processes?*

```
pid = fork();
if (pid == 0) {
    fork();
    ...
} else {
    ...
}
```

```
pid = fork();
if (pid == 0) {
    ...
} else {
    fork();
    ...
}
```

Now, write a C program that maps the LED matrix framebuffer device into its memory space (for example using `open_led_matrix()` in `led_matrix.h`), and then creates a child process using `fork()`. After the fork, the parent should

print

```
I'm the parent!  Lighting LED at (row, col)...
```

for some suitable row and column numbers `row` and `col` that you pick, and also light that LED in the matrix with some suitable color. The child process should instead print

```
I'm the child!  Lighting LED at (row, col)...
```

for some other values of `row` and `col` and also light that LED. The parent should `wait` for the child to terminate, and then sleep for two seconds before turning off all the LEDs (for example using `clear_leds()` in `led_matrix.h`. To sleep you can use the function `usleep` defined in the standard library header file `unistd.h`. Before the parent process terminates, it should unmap the framebuffer device and close it. If you opened it with `open_led_matrix()`, then you can simply do this with `close_led_matrix()`.

```
...
wait(NULL);
usleep(2000000);
clear_leds();
close_led_matrix();
...
```

Your program should properly check for errors from all functions that possibly return error values, and exit gracefully in these cases.

Also write a `Makefile` for your program and pack all the files (`.c` files, `.h` files and `Makefile` into a `.tar` archive file called `program-1.tar`.

```
tar -cvf program-1.tar your_program_directory/
```

You should upload this `.tar` file to the Student Portal along with your finished report. Also put the contents of your main `.c` file (not the `led_matrix` files or the `Makefile`) as a code listing directly into your report. For this, use the `lstlisting` LaTeX environment as shown in the template `.tex` file.

Listing 1.3 —— *Your* `.c` *file for the program above.*

4 *Task 2:* Processes and scheduling

For this task you will create processes that perform some (meaningless) computations to observe how they progress and how they are scheduled.

As a suitable piece of meaningless computation, you can use the following C function that takes about one second to complete when compiled by GCC with the default settings and executed on the Raspberry Pi.

```
/*
 * Do some pointless CPU-heavy computations. Takes about 1 second
 * to complete on a single core of the ARM Cortex A53 processor of
 * the Raspberry Pi 3 model B with default gcc 6.3.0 optimizations.
 */
void pointless_calculation() {
    int amount_of_pointlessness = 100000000;
    int x = 0;
    for (int i = 0; i < amount_of_pointlessness; i++) {
        x += i;
    }
}
```

Now, write a function `void run_child(int n)` that calls the above function eight times in a loop, and at the end of every loop iteration lights up a new LED on the `n`'th row of the LED matrix. That is, it first lights up the LED at position (`n`, 0), then the LED at position (`n`, 1) and so on, as in the following pseudo-code.

```
run_child(n) {
    row = n
    for col = 0 to 7 {
        call pointless_calculation() (takes about 1 second)
        light up led at position (row, col)
    }
}
```

Try the `run_child` function by calling it from the `main` function for some different values of `n`. The LEDs on the `n`'th row should light up one-by-one and the whole function take approximately eight seconds to complete.

When you are satisfied that `run_child` works as expected, rewrite the `main` function to create child processes that concurrently execute `run_child` for different values of `n`. First it should create one child process, then two, three and so on all the way to eight child processes. In between, the `main` function should clear the LED matrix. The child processes should terminate after finishing `run_child`, for example by calling `exit(0)`. The overall behavior of the main function should behave as the following pseudo-code.

```
open LED matrix
for num_children = 1 to 8 {
    for n = 0 to num_children - 1 {
        fork a new child process
        the child process executes run_child(n) and then terminates
             using exit(0)
    }

    wait for all the children to terminate (loop needed)
    sleep for one second
    clear LED matrix
}
close LED matrix
```

Question 2.1 —— *Describe the progress of the child processes that you can observe on the LED matrix, for the different number of processes that run concurrently. Explain clearly why they behave in this way considering the default scheduling and the number of cores in the Raspberry Pi's processor.*

Now make a minor change to the program above by having the child processes change their scheduling priority (i.e., their "nice" value) before executing `run_child`. Let the `n`'th child add `n` to its nice value by calling `nice(n)`, as in the following pseudo-code.

```
open LED matrix
for num_children = 1 to 8 {
    for n = 0 to num_children - 1 {
        fork a new child process
        the child process calls nice(n), then executes run_child(n)
             and then terminates using exit(0)
    }

    wait for all the children to terminate (loop needed)
    sleep for one second
    clear LED matrix
}
close LED matrix
```

Question 2.2 —— *Describe the progress of the child processes that you can now observe on the LED matrix, for the different number of processes that run concurrently. Explain clearly why they behave in this way considering the nice-values and the number of cores in the Raspberry Pi's processor.*

Write a `Makefile` for your program and pack all the relevant files into a .tar archive called `program-2.tar`. Upload this `.tar` file to the Student Portal as you upload the report. Also include the contents of your `.c` file as a code listing in the report.

```
tar -cvf program-2.tar your_program_directory/
```

Listing 2.2 —— *Your* `.c` *file for the program above.*

5 *Task 3:* Write a simple shell

In this task you will write your very own simple shell (like, for example, `bash`) that can be used to start other programs. Use the basic fork-exec pattern for launching new programs. That is, `fork()` a new child process and then replace its contents with another program using, for example, the `execl()` function from the exec family of system calls.

Remember that `execl()` must be given `(char *) NULL` as its final argument. Executing the `/bin/ls` program with no additional command-line arguments will therefore require a call like `execl("/bin/ls", "ls", (char *) NULL)`. See the `man 3 exec` for more details.

Your program should behave as follows.

1. Display a suitable prompt, for example "`myshell > `", and wait for input from standard input (i.e., the keyboard).
2. On input, if it matches a known command, continue to 3, else go to 5. If the command read from standard input is `quit`, exit the shell.
3. Fork a child process and exec the program corresponding to the command.
4. The main (parent) process waits for the child process to terminate, then go to 1.
5. Print "`Command unknown`" and go to 1.

Supported commands should include, at least, the following.

Input command	Program to execute
`ls`	`/bin/ls`
`pstree`	`/usr/bin/pstree`
`htop`	`/usr/bin/htop`
`ifconfig`	`/sbin/ifconfig`

Your program does not need to handle command-line arguments (like the "`-p`" in "`pstree -p`" or the "`-l`" and "`/dev`" in "`ls -l /dev`").

An example run of the program could look as follows.

```
myshell > ls
a.out   myshell   myshell.c
myshell > ifconfig
eth0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        ether b8:27:eb:3b:56:30  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

```
wlan0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.10.121  netmask 255.255.255.0  broadcast
            10.0.10.255
        inet6 fe80::a372:52f9:3ee1:1a2b  prefixlen 64  scopeid 0x20
            <link>
        ether b8:27:eb:6e:03:65  txqueuelen 1000  (Ethernet)
        RX packets 7399  bytes 1021520 (997.5 KiB)
        RX errors 0  dropped 2  overruns 0  frame 0
        TX packets 2861  bytes 664447 (648.8 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

myshell > quit
```

Upload the source file(s) for the above program to the Student Portal when you upload the report. If you have more than one source file, pack them into a `.tar` archive called `program-3.tar`. Include code listing(s) of your source file(s) in the report.

Listing 3.1 —— *Your source file(s) for the program above.*

6 *Stretch goal:* Write a full-fledged shell

(This task is completely optional. You don't need to include it in your report.)

Make your shell more full-fledged by adding (for example) the following features.

- Support for running any installed program, not just from a predefined list of commands. For this, `execlp` is probably helpful.
- Support for command-line arguments (e.g, the "`-p`" in "`pstree -p`").
- Support for running programs in the background by typing `&` at the end of the command line.

There are of course many more features that can be added. "Real" shells (`sh`, `bash`, `csh`, `ash`, ...) are often quite complex, and usually contain their own Turing-complete scripting languages. This seems to be over the top, but feel free to extend it in any way that seems interesting.

```
myshell > cowsay This is a really fancy shell!
 -------------------------------
< This is a really fancy shell! >
 -------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
myshell >
```