TECHNISCHE UNIVERSITÄT
CHEMNITZ

Professur
Betriebssysteme

Assignment 4

SWES WS 15/16

Dr. Peter Tröger

General Rules

All five assignments in the course sum up to 50 points, plus several extra points. You need to have at least 25 points to be allowed to enter the exam.

The solutions have to be submitted at:

`https://osg.informatik.tu-chemnitz.de/submit/`

Re-uploads are possible until the shown deadline.

For written tasks:

- Your submitted solution must be PDF file.

For coding tasks:

- Your submitted solution must be a ZIP / TAR archive, containing only source code files and a Makefile.
- After being uncompressed, your solution must compile when *make* is called in your solution directory.
- The resulting binary must have exactly the name given in the task description.
- There should be no sub-directories in your archive, or being created during compilation.
- You program must be executable without entering any keyboard input.

The submission system will give you automated feedback. If something is wrong, e.g. your code does not compile, or if your grading is done, then you will be informed via email.

Tutorials for C[1], Ada [2] and *make*[3] are available online.

[1] `http://www.tutorialspoint.com/cprogramming/`
[2] `http://www.infres.enst.fr/~pautet/Ada95/intro.htm`
[3] `http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/`

Task 1 (3 points)

The following C program is intended to compute the factorial of a number given as command-line argument. This is a working program, although some compilers may show warnings.

Find at least six mistakes that relate to C code quality / MISRA-C rules discussed in the lecture. Explain the problem and propose a correction for each one.

Submit a written solution as PDF file.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int input;
5
6  long int fac (long int input) {
7      switch (input)
8      {
9          case 0:
10             break;
11             return 1;
12         case 1:
13             return 1;
14         default:
15             return input * fac(input-1);
16     }
17 }
18
19 int main (int argc, char** argv)
20 {
21     long int x = fac(strtol(argv[1], NULL, 10));
22     if (x & 1 == 0) {
23         printf ("Uneven result: %ld\n", x);
24     } else {
25         printf ("Even result: %ld\n", x);
26     }
27 }
```

Task 2 (3 points)

Write an ADA program for a desktop machine that implements two tasks playing ping-pong with each other:

- Your program should have at least two active tasks during execution.
- Both tasks must synchronize their activities with the Ada rendezvous mechanisms:
  - Step 1: Task 1 starts by printing *ping* on the screen.
  - Step 2: Task 2 prints *pong* on the next line, but only after task 1 is finished with printing.
  - Step 3: Task 1 prints *ping* again on the next line, but only after task 2 is finished with printing.
  - Repeat from step 2
- The program should stop after 20 lines of output.

You program must be written in a way that it is easily extensible for more participating tasks (e.g. ping-pong-pang). The usage of Ada libraries is only allowed for doing the screen output.

Your Makefile should use *gnatmake* to generate an executable named *pingpong*. Submit your code as described on the first page.

Task 3 (3 points)

Draw a ladder diagram for the following PLC system:

- The system implements a fire alarm functionality with 4 smoke sensors. The sensors are connected to input port 1 - 4 of the PLC.
- The first level of alarm is a red signal light (output 17). It should be on while at least two of the 4 smoke sensors are active.
- The second level of alarm is a sirene (output 18). It should be on while at least three of the 4 smoke sensors are active.
- The third level of alarm is a fire department notification (output 19), which should only be active when all 4 sensors are active.

Submit your solution as PDF file.

Task 4 (5 points)

Write a C program that visualizes the current CPU load of the RaspberryPi computer on the Gertboard LEDs. You are allowed to use the *wiringPi* library for accessing the hardware.

On full load of the CPU, 10 LEDS should be switched on. On zero load of the CPU, no LED should be switched on. Partial loads should be shown accordingly with partial LED lighting, for example, 50% load should be visualized by activating only 5 adjacent LEDs, starting from B3. Your program should constantly monitor and display the CPU status for 10s and terminate then. The display should update at least every second.

Your Makefile must use the cross-compiler named *arm-linux-gnueabihf-gcc*. The compiled binary must get the name *cpuload*, and must be executable on a RaspberryPi computer. Submit your code as described on the first page.

Before using the lab hardware, make sure that the Gertboard is wired correctly, as shown below.

One way to determine the CPU load is reading the file `/proc/stat` in a loop. On the Raspberry Pi devices, the first line of the file looks like this:

```
cpu  8494 240 1808 325873 13397 0 22 0 0 0
```

Each value represents a particular statistic, as sum for all (!) CPU cores in the system[4]. The fourth value represents the time spent in idling since system start, accumulated for all cores. This value can can be used to determine the time fraction having computational load, by considering the round length for the measurement cycle.

Since RaspberryPi Linux and standard desktop Linux are nearly identical, you can test your load computation approach on a desktop machine, before porting it to the embedded device. Use the *top* command as reference. If you need to produce arbitrary CPU load, run the *find / command* in the background.

[4] `http://man7.org/linux/man-pages/man5/proc.5.html`

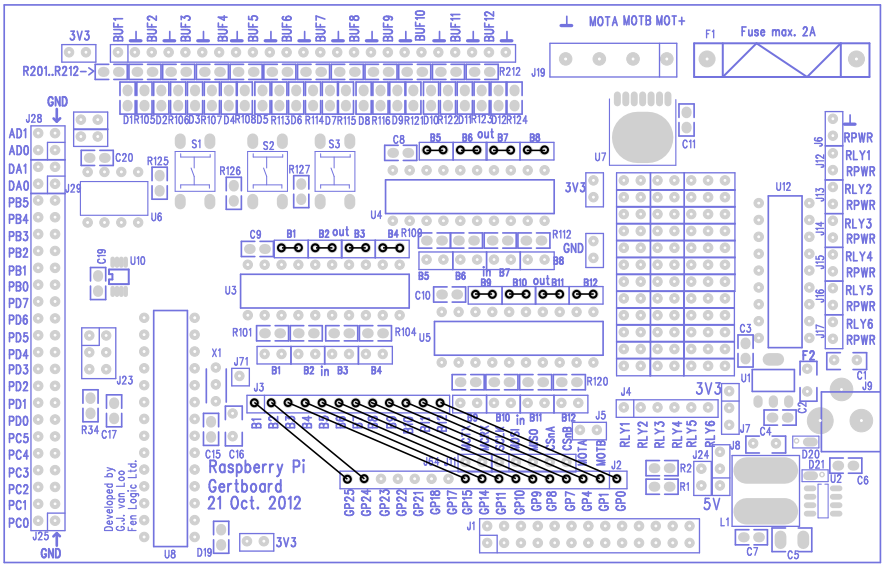

Figure 1: Schematic of the experiment set-up

Task 5 (3 extra points)

Extend your program from Task 4 so that the push buttons on the Gertboard change the operation mode:

- Pressing S1 (GP25) changes the program to sample the CPU load every 20ms.
- Pressing S2 (GP24) changes the program to sample the CPU load every second (default).

Please note that these buttons need to be pulled-up[5].

Task 6 (5 extra points)

Write a cross-language remote control, where a program on your desktop machine is controlling the Gertboard LEDs. This works by having a client application on the desktop machine, and a server application running on the RaspberryPi.

The client software on the desktop computer must be written in Ada. This software should:

- Open a socket and connect to a server port on the RaspberryPi.
- Send a 16 Bit value that comes from `/dev/urandom`, in a loop with some delay.
- Terminate after 10 seconds.

The server software on the RaspberryPi must be written in C. This software should:

- Open a server socket and wait for incoming connections.

[5] `http://bit.ly/224pHQ7`

- Receive a 16 Bit value and use the lower 10 Bits to activate / deactivate the LEDs accordingly. When reading the next value, the previous one should be overwritten.
- The program should terminate when the client application disconnected.

Please note that everything beside the hardware control part can be tested on the desktop machine. Structure your software accordingly. Your Makefile should produce two binaries in one run:

- The C program for the RaspberryPi should be compiled with the *arm-linux-gnueabihf-gcc* cross compiler, resulting in a binary named *server*.
- The Ada program for the desktop machine should be compiled with *gnatmake*, resulting in a binary named *client*.