Exercise Sheet 5

Exercise 1 (Interrupts)

1. What are interrupts?
2. What is the interrupt vector?
3. What are exceptions?
4. What happens, if during the handling of an interrupt, an additional interrupt occurs?

Exercise 2 (Programs and Processes)

In this exercise you will investigate the content of a program and its memory layout as a process.

1. Create a file that contains at least one C function but **no** `main()` function. Create another file that contains a `main()` function which calls at least one of the functions from the other file.
   Compile both files using `gcc`, passing the parameters `-g` and `-c`. What do these parameters do and what kind of files are created? Use the command `nm` on these files and interpret the output.

2. Invoke the linker as follows:
   ```
   gcc -Wl,-start-group file1.o file2.o -Wl,-end-group -g -o
   <outputfile>
   ```
   Use the tools `ldd` and `nm` on the resulting binary and understand the output. Invoke the linker as before but additionally pass the parameter `-static`. Call `ldd` and `nm` again and check the difference.

3. Debug the program using `gdb`. Use the `gdb` commands `breakpoint`, `print`, and `backtrace` from the `gdb` CLI. What are the addresses of the variables and functions in your program?
   Hint: You can use the CLI command `tui enable` or invoke `gdb` with `-tui` for an text user interface.

Exercise 3 (Building your own Kernel)

In this exercise you will build your own kernel and boot it in an emulator environment.

1. Install the required packages via your packet manager. On Debian Linux (which is installed on the lab computers), you can use the following command:
   ```
   sudo apt-get update && sudo apt-get install git build-essential
   bc kmod cpio flex libncurses5-dev libelf-dev libssl-dev dwarves
   bison initramfs-tools
   ```

2. Obtain the Linux kernel source code from the git repository via
   ```
   git clone https//git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
   ```

3. Create the default configuration via calling `make defconfig` (from within the cloned kernel repository). Now edit the configuration by calling `make menuconfig`. Append a custom string to the kernel version string and set the default hostname to a custom value.

4. Build the kernel via `make`.
   Hint: To speed up the process you can use the parameter `-j` to run multiple

(compiler) jobs in parallel. The optimal value for the argument can be found with the command `nproc`.

5. Create a `ramdisk` using the command `mkinitramfs`.

6. Install the emulator `QEMU` by calling:
   ```
   sudo apt install qemu-utils qemu-system-x86 qemu-system-gui
   ```

7. Boot your custom kernel with your ram disk by calling:
   ```
   qemu-system-x86_64 -kernel arch/x86_64/boot/bzImage -initrd
   <path/to/your/ramdisk>
   ```
   Check the kernel and host name using `uname`.

8. Create your own `init` program and spawn it from your kernel in the emulator. Compile and link a static C program that ends with a long sleep, e.g., by calling `sleep(0xFFFFFFFF)`. Put the binary in an empty directory, change to that directory, and call:
   ```
   find .  | cpio -o -H newc | gzip > rootfs.cpio.gz
   ```
   Pass the generated rootfs.cpio.gz as a parameter to the qemu call (as an argument for `-initrd`).