To begin with, boot (or reactivate) the Ulix Devel VM and run the command in the shell update-ulix.sh out. This will download the files that you need to process the project tasks.

In the folder ~ / ulix / (/ home / ulix / ulix /) you will find the Ulix version 0.10 - this is not the reduced Ulix code previously known from the exercises, but the "real" Ulix code, which offers significantly more features than the versions from the exercises. The associated Literate Program
ulix-book.nw contains a chapter at the end (*chapter) Internship:* Here you add one for each task you have worked on *section* with the task number as a heading (\ section {task X}).

The file is an older version of the textbook *The Design and Implementation of the ULIX Operating System,* which you received as a PDF file at the beginning of the event and which describes the Ulix 0.13.

For many tasks it will be unavoidable to also adapt the existing code, for example to incorporate the "call" of an additional code chunk in the kernel initialization. If you do this, refer to the supplement at a suitable place in the internship chapter.

For all exercises on system calls, you can look up how the transfer of parameters and the return of the return value work in the old exercises - the principle is always the same (and it works in the "large" Ulix exactly as in the previously used small ver - sions).

If you do not succeed in solving a task, or not in full, you can still hand it in - then include information on the incorrect behavior and any assumptions about the causes of the error in the literate programming documentation.

Important:

1. You can while solving the tasks *with restrictions* **work together.** If you want to form a group of two, you should develop while working together on two separate computers and then only discuss questions of detail ("How did you solve XYZ?"). You can also ask me for help during the on-site appointments or after the appointments by email if you get stuck at a point.

2. **The Literate Program (** which includes code and documentation) **each participant has to make it individually.** Identical or structurally identical documentation in the submissions of two or more participants is a non-refutable indication of unauthorized cooperation and leads to failure.

3. When programming, please take into account that **50% of the achievable points for the quality of the documentation in the literate programming style** be awarded. So don't create the program first and then add documentation - this approach usually leads to poor LP style and attracts attention.

   A cleanly written LP-style program with minor errors in the code will typically receive a better rating than a program with perfect code and documentation that has been added later.

Notes on the various types of tasks follow.

Compulsory and optional tasks

There are two types of tasks:

- The first tasks **P1** - **P3** are compulsory tasks, they essentially serve to get started with the job. For the **P-** There are tasks as a whole **20 points.**
- This is followed by elective tasks **W1** - **W6.** Here you can select multiple tasks as you wish, so that you can click on in the W area **60 processed points** come. Each task has a number of points after the title. Tasks that consist of several subtasks (a, b, c, ...) can only be selected completely.

  Example: You work on W1, W2 (10 points each) and W5 (40 points), total 60. Or: You work on W5 and W7 (40 + 20 = 60 points).

  It is *Not* possible to do more elective tasks and solve all of them only partially - you have to make a choice.

  The tasks W5 to W7 marked with two asterisks are somewhat more demanding than the other tasks.

So a total of 20 + 60 = 80 points can be achieved. (If you make a mistake in your selection and work on too many tasks, I will ask you by email which ones you want in the evaluation.)

Submission, deadline

Send one Zip or tar.gz- Archive with the files ulix-book.nw, ulix-book.pdf and all self-created program files lib-build / tools / by email to me. When evaluating literate programming, I only consider parts of the text that are in the last chapter (*Appendix C: Internship,* from page 431). Deadline for the submission is on Sunday, **07/02/2015 (** 11:59 p.m.).

required tasks

P 1. Overview of source code and build process (0 points)

To begin with, you will familiarize yourself with the directory structure of the Ulix source code and the build process. There are no points for this task, it is just a guide.

Right in ~ / ulix / (/ home / ulix / ulix /) is the most important file, the Literate Program ulixbook.nw. If you switch to this folder, you can use the automatic translation make
and running ulix with make run toast (as in the exercises). about make pdf generate the formatted documentation (ulix-book.pdf). With around 450 pages, it is about ten times as long as the documentation seen in the exercises so far. It is not necessary to read all of these pages, but you should get an overview of the structure so that you can quickly find relevant program parts - in the source file or in the PDF file.

The quickest way to get an overview of the code is to look at the PDF file - in it you can click on Code Chunks to get to the respective code positions. The definition of the source code file bin-build / ulix.c begins on page 25 in the code chunk < *ulix.c>,*
the initialization of the kernel ends with the change to user mode and the start of the shell (in < *start shell>).* You do not have to write assembly code as part of the project; if you want to peek inside, you can find it in the file bin-build / start.asm.

All Ulix applications use a user mode library, which is implemented in Chapter 21 (from page 405). It is only very incompletely documented and does not yet offer too many features; you will add some as part of the project tasks. There are only two code chunks < *ulixlib.c>* and < *ulixlib.h>,* which at make- Call to the files lib-build / ulixlib.c and

lib-build / ulixlib.h extracted.

Ulix programs can be found in the subfolder lib-build / tools. If you want to create a new user mode program, create a new C file in this folder - at make- When it is called, all C programs in this folder are automatically compiled and the resulting program files end up in the floppy disk image that Ulix uses as the root file system. If you create such a program as part of a project, you have to copy the code from the C program by hand ulix-book.nw copy. The floppy image mentioned is binbuild / minixdata.img. To copy new folders or files to the image, you can use them in the lib-build / diskfiles put down - on the next run of make the files then end up in the image. If you want to delete files from the image, you can do this under Ulix with the rm- Complete the order.

With the Ulix system running, take a look at the folder am / and try out some of the programs there. cat, clear, cp, diff, hexdump, ps, rm, touch and vi work in a similar way to the corresponding Linux applications, the other tools have no meaningful task and are intended for testing Ulix features.

P 2. Hello World (12 points)

In this exercise, you essentially repeat work steps that you already know from the exercises; the point here is to get used to the new and more complex code environment. The aim is to build a new system call into the system and call it up from the main program.

In contrast to the Ulix versions in practice mode, the correct Ulix has a file system that is in the folder am / contains executable Ulix programs. The process of translation and inclusion in the file system image is automated by the Makefile. (So you no longer have to, as in exercise

12–14, manually insert lists of hex numbers into the Ulix source code.)

a) Write a kernel function

void u_hello (int n)

the " Hello World -% d \ n " and the argument via the format string n issues. You take the prototype in the code chunk < *function prototypes>* on, the implementation in < *function implementations>.*

b) Write a syscall handler

void syscall_hello (context_t * r)

which from r extracts an integer argument and thus the function u_hello () calls. The syscall handler also needs a prototype and implementation in the appropriate code chunks.

c) Define a syscall number for the syscall, e.g. B. with

# define __NR_hello 777

and carry with you install_syscall_handler () Enter the new syscall in the syscall table. You can make the call in the code chunk < *initialize syscalls>* record, tape.

d) Write a user mode library function void hello (int n), which via sys-
call2 () the Syscall 777 executes. Library functions also require prototypes and implementation, but other code chunks are responsible for this.
The prototype belongs to < *ulixlib.h>,* the implementation according to < *ulixlib.c>.*

e) Translate with make the modified code to make sure it compiles correctly.

f) Change to the subfolder lib-build / tools / and change the program file that already exists there hello.c from: In the Main()- Test function Hello (), z. B. With hello (5); the remaining commands in Main() remove. Then switch back to the main folder (to / home / ulix / ulix) - if you are now make the new program will be compiled. Start with make run the system and test the command Hello. The program Hello is (under Ulix) in the folder am /.

(It is important in all Ulix programs that you copy the header file ../ ulixlib.h With

# include "ulixlib.h"

Include and - if you next Main() implement other functions in the program, always with the code of Main() start. Then you have to declare the other functions in advance by specifying the prototype. You can find an example of this in diff.c: There will memcmp () declared in advance and only after Main() implemented. In addition, every Ulix program must explicitly include exit (0) be terminated - an exit from Main() With return causes error messages in Ulix.)

P 3. Status with Ctrl-Esc (8 points)

If you press [Shift-Esc] while Ulix is running, you will land in the kernel shell (and all processes will be stopped). Give there exit on, you return to User Mode and the processes continue.

The switch to the kernel shell happens in the keyboard interrupt handler keyboard_handler () (from page 183). Take a look at how the handler responds to [Shift-Esc].

Now insert a call to the Code Chunk <at a suitable point in the code *project keyboard dealer>* one. You can implement this code chunk in the project chapter. It should be analogous to the

Call the kernel shell a function void status_break () when you press [Ctrl-Esc].

This now has to be implemented. Ulix always shows a status line at the bottom anyway. The function you want to write void status_break () is designed to perform the following tasks:

- Deactivate interrupts with < *disable interrupts*>
- Build an information string with a maximum of 80 characters, which contains the current thread ID, the current address space ID, the free frames, the number of all threads, the number of all threads in the ready queue, the Ulix version number.
- this string with set_statusline () (see p. 381) display wait approx. one
- second (make a suitable counting loop) status line back to standard
- content (UNAME) set interrupt reactivate with < *enable interrupts*>
-

Elective tasks

*Select here several tasks that together **60 points** are worth.*

*The tasks W 5 ** to W 7 ** are somewhat more difficult than the others and are therefore marked with **.*

W 1. More shells (10 points)

Ulix uses ten virtual consoles, which you can reach with [Alt-1] to [Alt-9] and [Alt-0] - the first five consoles run shells. Adjust the Ulix code so that eight shells (on the first eight consoles) are available. First find out where (in the kernel, in the library?) The consoles are activated.

To document your changes, add the *Project-* Chapter new code chunks (with your assigned chunk names) which contain the changed one. In the original places in the code, delete the original code and reference the new code chunk.

W 2. Syslog in the file system (10 points)

This task is coming *without* System calls off. Write two new kernel functions

```
void syslog_open (char * filename); void syslog_write
(char * msg);
```

with which you can open a syslog file and write strings into it. It should
syslog_write the transferred string always has the current value of system_ticks and prepend a space and a line break character (\ n) append at the end. You can use the kernel function to format a string sprintf () that works like the Linux function of the same name.

Then find a kernel function that is called frequently and add a call to the form there

```
syslog_write ("function name: entering");
```

at the beginning of the function.

At a suitable point (during the initialization of the kernel) you must then syslog_open ()
call. If there is a call to syslog_write () comes before the log file has been opened, the function should simply include return to return.

In the newly compiled system, test whether you can enter multiple cat log file name can see how new entries are added.

W 3. Boost: Only execute this process (20 points)

The scheduler used by timer_handler () is called every two ticks, it switches to the next process in the ready queue. In this task, you give a process the option of preventing this switchover: Using the function void boost (int n) he can set a global variable (global in the Ulix kernel), and in the timer_handler (in one of the < *timer tasks>)*

should then be checked whether this has a value! = 0. If so, the value is decreased by 1 and there is no context switch.

a) Declare a global variable at a suitable point int boost_count, which you set to 0 in-
initialize.

b) Move the code chunk < *timer tasks>,* of the scheduler () calls for the practical
cum appendix so that you can document the changes required there; it's about this chunk:

```
< timer tasks> =
  // Every 5 clocks call the scheduler if (system_ticks% 5
  == 0) {
    [...]
        scheduler (r, SCHED_SRC_TIMER); // defined in the process chapter [...]
```

c) Adjust the case distinction; it must also include the condition boost_count == 0 he-
be full. You also need one else- Case you boost_count decrement.

d) Write a syscall handler

```
void syscall_boost (context_t * r);
```

who reads the transferred value from the correct register and checks it boost_count writes (if it is positive or 0). Define a Sycall number constant __ NR_boost (using a number that has not yet been assigned) and enter the correct code chunk
a call to install_syscall_handler () one.

e) Write a user mode library function

```
void boost (int n);
```

which with the help of syscall2 () triggers the syscall. You can take a look at the implementation of open () orientate.

f) Write a small user-mode program that you can use to see the effect of boost () about-
can check.

(In this exercise we deviate from the standard procedure: Normally there would still be a kernel function u_boost, which would be called by the syscall handler - but since this is simple

```
void boost (int n) {boost_count = n; }
```

it makes no sense to turn it into a function.

W 4. Unix name: uname () (20 points)

All Unix systems bring a command line program uname with, which gives information about the running operating system. The program uses a system call with the same name for this.

a) Read the man page for the Linux library function uname by (man 2 uname; without the 2 you get the manual page for the shell command uname), Pay particular attention to the data structure struct utsname.

b) Define a new type in Ulix struct utsname analogous to the definition from the Manpage, but leave the entry char domainname [] path.

c) Implement a new function u_uname ():

```
int u_uname (struct utsname * buf);
```

which the transferred buffer with suitable values (sysname: Ulix, node name: ulixmachine, release: 0.10 or last four characters in UNAME- # define- Constant, version: BUILDDATE- # define- Constant, machine: i386) fills.

d) Write a syscall handler syscall_uname () with the following signature:

```
void syscall_uname (context_t * r);
```

He is supposed to function u_uname () call.

Also remember to register the new syscall handler. The syscall number is in ulix-book.nw already in the line

```
# define __NR_uname                    122
```

declared.

e) Add the type declaration of. Already used in the kernel to the user mode library struct utsname as well as a function

```
int uname (struct utsname * buf);
```

which with the help of syscall2 () triggers the syscall. You can take a look at the implementation of open () orientate.

f) Develop a new user mode program uname (in the folder lib-build / tools /), the the uname- Function and the results in a similar form to that uname- Program under Linux (when called with option - a) issues. Test that the program starts under Ulix and that it outputs the correct information.

W 5 **. Ridiculously Simple Filesystem (40 points)

The Ridiculously Simple File System (RSF) works with a standard sector size of 512 bytes, and an image has the following structure:

- Sector 0 contains the FAT (File Allocation Table), which contains up to 32 FAT entries with a size of 16 bytes (16 x 32 = 512).
- Each FAT entry contains the following data:
  - Bytes 0–11: file name. If the file name is less than 12 characters, the remaining bytes contain \ 0. (Attention: With file names of length 12 there is no \ 0- Termination of the name.)
  - Bytes 12–13: file size in bytes, so the maximum file size on an RSF medium is 2 ^{16} Byte = 64 KByte.

    ◦ Bytes 14–15: Sector number of the first data block of the file

- All files have to be saved together in the image because only the start sector is noted in the FAT. The first file starts in sector 1, just after the FAT. You can recognize unused FAT entries by the fact that the first byte of the entry \ 0 is. An unused entry may not be followed by any used entries.

a) Develop a small (Linux) command line tool mkfs.rfs, that you in the form
. /mkfs.rfs disk.img file1 file2 file3 ... call to the files file1, file2, fi
le3, ... into the image disk.img to copy. All source files must be in the current directory (the tool cannot deal with path information); the target file (in the example disk.img)
but may also be in a different folder.

b) You will now add an RSF driver to Ulix. To do this, develop the following func-
functions that all assume that the second virtual disk (hdb.img) a with mkfs.rsf The generated RSF image contains:

- void rsf_ls (): Outputs a list of all files as follows (size in bytes)

```
filename              size sectors
test.txt                30 0001-0001
test2.txt            32768 0002-0065
Makefile                12 0066-0066
```

- int rsf_open (char * filename): a simplified version of file opening. A maximum of one file can always be open. Reacts depending on the situation rsf_open as follows:
    - A file is already open → cancel, return value -1.
    - No file is open yet, but filename does not exist on the data carrier
      → Abort, return value -1.
    - No file is open yet, and filename exists → success. The function notes the name of the file in a global variable. Return value 0.
- int rsf_readsector (int fd, int secno, char * buf): reads a sector from the
  opened file (if fd == 0). If no file is open or fd! = 0 applies, termination with return value -1. Otherwise: return value = number of bytes read. Caution: For the last sector of a file, this is often a value <512.

- void rsf_close (int fd): closes the file (if one was open); fd is arbitrary here.

There is no writing function because RSF is a read-only file system, similar to the ISO file system on CDs and DVDs.

Note: In order for Ulix to have access to a second disk image, you need to enter the bin-build / Makefile in the target run (in the last line) the call to qemu adapt, you need the additional option - hdb hdb.img (if the image hdb.img is called and also in bin-build / lies).

c) Add system calls and user mode library functions that you can use rsf_open, rsf_ readsector and rsf_close can use in user-mode programs and write a test program.

d) Based on rsf_readsector () now write functions rsf_lseek () and rsf_ read (), the like lseek and read work (see manpages); When doing this, attention must be paid to reading areas that exceed block boundaries. In a final step, you can integrate the RSF file system into the VFS from Ulix, ie mount an RSF data carrier and transfer it regularly u_open, u_read etc. use: Customize the u_- Functions of the VFS so that in addition to the Minix functions mx_ * now also the new RSF functions rsf_ * be called.

W 6 **. Process prioritization (20 p.)

Under Ulix, all processes or threads are equivalent, there is no prioritization. In this task, you implement priorities.

a) Add a "nice value" to the thread control block (int nice), of values between -20
and 19 can accept. The default value is 0. Processes inherit via fork () their nice value to child processes.

b) Write a function int u_setpriority (int nice), the
Nice value changes to the specified value. Add suitable functions syscall_setpriority in the kernel as well as int setpriority (int nice) in the user mode library. (The return value of u_setpriority or. setpriority is the new nice value. The signature does not match that of setpriority () under Linux, a look at the man page under Linux is still helpful.

c) Adjust in the code chunk < *timer tasks>* the code block

```
if (system_ticks% 2 == 0) {
    [...]
    scheduler (r, SCHED_SRC_TIMER); // defined in the process chapter [...]
```

so that the process is not simply changed with every second timer interrupt, but that the Nice value of the current process (thread_table [current_task] .nice)
is taken into account - as a result, a process with a smaller Nice value should receive a longer time slice than one with a larger Nice value.

d) Write a test program with which you can check that the nice values really are
be taken into account by the system.

W 7 **. atexit (): Configure the end of the program (20 p.)

Unix systems give processes the ability to use atexit () specify a function to be used when exiting the program with exit () or return (out Main() out) is called. Implement atexit () under Ulix. To do this, you have to add a new field to the thread control block that can store the entry address of a function. Notes on the function of

atexit () you can refer to the Linux man page too atexit remove.

As part of this task, you need a kernel function u_atexit (), a syscall handler
syscall_atexit () as well as a user mode library function atexit () write and the radio
tion syscall_exit () to adjust.

An alternative option is the user mode library function exit () in _ exit () renamed and a new function exit () to be included in the library, which may have been registered first atexit- Function and then _ exit () calls.

Under Linux & Co. it is possible atexit () repeatedly, at the end of the program several registered functions are called. That is not necessary for this task: One atexit- Function is enough.

Write a small test program that you can use to test your implementation.

Good luck and above all: HAVE FUN!