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 you need to edit the current
len practice exercises.

12. User mode programs

In the folder tutorial06 / a version of the Ulix kernel can be found in your home directory, in
which the new code for switching to user mode was implemented. It lies as a literary program (ulix.nw) which includes the sample solution for the keyboard interrupt handler.

In this task you will get your first user mode application up and running.

As usual: when programming, make sure you have a *Literate Program* create, i.e. integrate code and documentation well into the overall document.

a) Test program: First, create a small test program for ULIX so that you can see
where the journey is going. The actual main program Main() in the file test.c should just off
consist of the following lines:

```
int main () {
    printf ("Hello - User Mode! \ n"); for (;;);
                              // infinite loop
}
```

The function printf () you still have to define it yourself: tie one of the syscall * -
Features (syscall1, syscall2 etc.) into the file. Our simple printf- Variant only accepts a single string argument, so it works like the function userprint () from the last exercise. The constant __ NR_printf (1) you also have to start with # define define.

Important: In our user mode programs, the Main- Function always comes first. Features you made Main() call out (e.g. printf) do you have to *above* from Main() pre-declare (prototype), but *below* from Main() to implement! The
is due to the fact that ULIX later loads the generated program to virtual address 0 (and following) and execution also begins at address 0 - the code of the main program (Main)
must therefore be at the top of the generated program file.

You already need the necessary for compiling with the compiler gcc- The call is entered in the Makefile, you don't have to change anything:

```
    $ (CC) -nostdlib -ffreestanding -fforce-addr -fomit-frame-pointer \
- fno-function-cse -nostartfiles -mtune = i386 -momit-leaf-frame-pointer \
- T process.ld -static -o test test.c
```

That creates (if you make enter) from the source code in test.c the executable program
test.

b) Install disassembler: For this task you need a disassembler that consists of a
binary program file makes readable assembler source code again. If you entered x86dis receive an error message ("Command not found"), install the
Disassembler with the following command:

```
sudo apt-get install x86dis
```

c) Disassemble the generated program test with the command

```
x86dis -e 0 -s intel <test | sort -u
```

The output should start as follows:

```
00000000 8D 4C 24 04                         lea     ecx, [esp + 0x4]
00000004 83 E4 F0 00000007                   other   esp, 0xF0
FF 71 FC 0000000A 51                         push    [ecx-0x4]
                                             push    ecx
0000000B 83 EC 08                            sub     esp, 0x08
0000000E 83 EC 0C                            sub     esp, 0x0C
00000011 68 A2 00 00 00                      push    0x000000A2
00000016 E8 77 00 00 00                      call    0x00000092
. . .
```

That is the beginning of the translated Main()- Function; of the call Command calls the printf () - Function on.

d) Since we don't have a filesystem to copy the program to, we'll use one
Trick: We write the code directly into the kernel and later copy it into the user mode memory. The tool helps here hexdump, whose output format is based on a format
string lets you set:

```
hexdump -e '8/1 "0x% 02X,"' -e '8/1 "" "\ n"' test
```

This produces output of the following form:

```
0x8D, 0x4C, 0x24, 0x04, 0x83, 0xE4, 0xF0, 0xFF, 0x71, 0xFC, 0x51,
0x83, 0xEC, 0x08, 0x83, 0xEC, 0x0C, 0x68, 0xA2, 0x00, 0x00, 0x00,
0xE8, 0x77, [. ..]

0x6F, 0x64, 0x65, 0x21, 0x0A, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00,
*
```

Compare these hex numbers with the hex numbers output by the disassembler: They are identical.

You can convert this output (without the line with the asterisk) into the ULIX source code (in < *global variables>)* take. Declare in **Chapter 12.6** usermodeprog in the shape

```
unsigned char usermodeprog [] = {
    0x8D, 0x4C, 0x24, 0x04, 0x83, 0xE4, 0xF0, 0xFF, 0x71, 0xFC, 0x51,
    0x83, 0xEC, 0x08, 0x83, 0xEC, 0x0C, 0x68, 0xA2, 0x00, 0x00, 0x00,
    0xE8, 0x77, [. ..]

};
```

As a result, the individual bytes end up in a suitable array. If there is a line at the end that contains only zeros (0x00) you can remove it as well.

You can now write a function that loads the program.

e) In the ULIX code in the chunk, add < *kernel main: user-defined tests>* at the end of the line

```
start_program_from_ram ((unsigned int) usermodeprog, sizeof (usermodeprog));
```

a: The function start_program_from_ram (which you still have to partially implement)
will load the program and start it, for which purpose it changes from kernel mode (ring 0) to user mode (ring 3).

13. Activate user mode

In this task, you implement some of the methods necessary to start a program in ULIX. Many parts of the code from the lecture are already integrated.

a) The most important function is start_program_from_ram (): It works similarly to the one in Function presented in the lecture start_program_from_disk (), that we don't use can - differences are: loading from memory (instead of from disk), no consideration of the scheduler (we only start a single process at first).

You don't need to type the following code; he's already in ulix.nw contain.

```
void start_program_from_ram (unsigned int address, int size) {
    addr_space_id as;
    thread_id tid;
    < start program from ram: prepare address space and TCB entry>
    < start program from ram: load binary>
    < start program from ram: create kernel stack>
    current_task = tid;                         // make this the current task // jump to user
    cpu_usermode (BINARY_LOAD_ADDRESS,
                  TOP_OF_USER_MODE_STACK);      mode
};
```

b) The following steps are necessary to implement the missing chunks:

Fill in (in < *start program from ram: prepare address space and TCB entry>,* **Chapter 12.4)** the two variables as and tid with content by using the features create_new_address_ space () and register_new_tcb () call. These functions work as shown in the lecture and are in the ulix.nw- File for today's exercise already included. Just pay attention to the call sequence: register_new_tcb () requires the address space ID as an argument. Give new processes 64 KByte program memory and 4 KByte user mode stack.

Further tasks for the first code chunk are to fill the thread control block elements with meaningful values. Assign the value 0 as the PPID (Parent Process ID) for the new (first) process.

The code chunk < *start program from ram: create kernel stack>* is largely with the code Chunk <shown in the lecture *start program from disk: create kernel stack>* identical; it is already included in the program.

The code chunk <is still missing *start program from ram: load binary>:* In this you have to memcpy () the program code from the array, whose address and length you can assign to the function in the parameters address and size copy it to address 0.

cpu_usermode () is an assembler function; You can find this in start.asm.

c) Test your code (with make and make run) - After the general greeting ("Hello World"), ULIX should also output the line from the User Mode program ("Hello - User Mode!").

14. More features for user mode

In this task you expand the features of the User Mode. The aim is that the main program in the user mode program with readline () Can read text from the keyboard. These are - in **chapter 12.7** - several steps necessary:

a) Write a system call handler that does the job kreadline () calls. Like all sys- call handler he has the signature

```
void syscall_readline (struct regs * r);
```

When called, contains r-> eax the syscall number (which you can ignore), and r-> ebx should contain the address of a string declared in the user mode program. To do this, make a reservation in the user mode program (test.c) a variable that can hold 256 characters. We don't pass the length; you can do this when you call kreadline () set to 256. In the handler, before calling kreadline () activate the interrupts, because by jumping to the handler they are deactivated. So add this line:

```
asm volatile ("sti");                    // before kreadline ()!
```

Assign a syscall number __ NR_readline and enter the handler in the Syscall table.

b) Now write in test.c a function void readline (char * s); which about a of the syscall * - Functions make the correct system call; you have to do this in the file test.c the constant __ NR_readline define (see exercise 12a). The same applies here again: The new function must be above Main() pre-declare and below Main() to implement.

c) Customize the function Main() in test.c so that it reads text in an endless loop and outputs again, e.g. B.

```
int main () {
   char s [256];
   printf ("Hello - User Mode! \ n"); for (;;) {

      printf (">"); readline (s);
      printf ("Input was:"); printf (s);
   }
}
```

d) Test your program. test.c you need to after compiling with make again as in Exercise 12 convert d) to e) into hex code and convert to the kernel code in ulix.nw integrate. Then you call make again to compile the customized kernel.

The output function now masters scrolling, so that you can continue testing when you arrive at the bottom of the screen. To understand the scrolling feature, look at the function scroll () and the two places where it is called.

e) The function scroll () determined (as well as kputch ()) the correct memory address, among other things by evaluating the variable VIDEO: Find the places in the code where VIDEO is changed - the variable takes on three different values during initialization (0xc00b8000, 0xb8000 and 0xd00b8000): Why is that? As a reminder: The physical memory addresses where you can find the text framebuffer of the graphics card start at 0xb8000.

f) **Additional task:** Apply the code in test.c into the Literate Program ulix.nw and adjust the Makefile so that the file test.c is extracted from this literate program. As a code chunk name for the code in test.c use < *test.c>.* You then need an additional call to notangle analogous to

```
notangle -Rulix.c ulix.nw> ulix.c
```

with the option - R. Use the new chunk name and also adjust the output redirection with>. Note that in the makefile all commands must be indented with [Tab].

Grasp code that is in both ulix.c as well as test.c identical occurs by suitable Code chunks together, which you incorporate in both files - then the previously double lines of code will only appear once in the Literate Program.