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.

9. Keyboard driver: polling and interrupts

In the folder tutorial04 / a version of the Ulix kernel can be found in your home directory, in
which the new code for interrupt and fault handling was implemented. It lies as a Literate Program (ulix.nw) in front. In this task, you will develop a keyboard driver. Pay attention to
Programming to get you a *Literate Program* create, i.e. integrate code and documentation well into the overall document.

a) Polling: Go to the folder and open the file ulix.nw. You will find in the end
a “Keyboard Drivers” section where you can place your new code (or most of it) - the rest of the file corresponds to the Literate Program from the last exercise, expanded by the mechanisms presented today.

In the first step you test how the keyboard query works with polling. To do this, you need some information:

(i) The keyboard controller has two ports (0x60 and 0x64), that you with inportb () can read out. Define in < *constants>* the port numbers:

```
# define KBD_DATA_PORT          0x60
# define KBD_STATUS_PORT 0x64
```

Information on pressed / released keys is available at the data port, and you can use the status port to check whether a key has been pressed at all.

(ii) First try to read out the data port again and again in a loop and output the result (as a number): You can answer the query with

```
unsigned char scancode;
scancode = inportb (KBD_DATA_PORT);
```

take care of. (Your code is an empty code chunk < *kernel main: user-defined tests>* at the end of the Noweb file, which is called after the initialization and the tests known from the last exercise.) Then enter the scan code printf () out.

That will happen even with one printf () - Call without "\ n " fill the screen relatively quickly. That's why you build in front of the printf () - Call an exam: if (posy == 25) clrscr ();

The command clears the screen when you reach the bottom of the screen and the output continues at the top.

You will find that this implementation produces a steady (and rapid) stream of data. While the system is running, press various keys; You will then see that the output changes. (You will likely have to hold down the key to see a change.) What it shows is a keyboard scan code; each value corresponds to a press or release. The same value appears over and over again until you press a key or release it.

(iii) You can improve your code by also querying the status register via the status port; it works like the data port, only with KBD_STATUS_PORT. If the returned
given value has the lowest bit set (which you can do with if (status & 1 == 1) test), there is a fresh scan code, and only then do you query the data port. In the changed

Version only appear when you press or release a key.

(iv) The scan codes for pressing and releasing a key differ only in the eighth bit that is set; for example, the scan codes for the “A” key are 30 and 158 (= 30 + 128), for the “S” to the right of it they are 31 and 159 (= 31 + 128). Create an assignment table that contains the ASCII value of the respective capital letter for some scan codes. You do not have to look up the ASCII values for this; B. ' A ' or ' B ' in the

Enter table. Initialize the table with zeros as follows:

```
char scancode_table [128] = {0};
```

You can then e.g. B. for the keys "A" and "S" whose scan codes (30, 158 or 31, 159) you already know,

```
scancode_table [30] = 'A';
scancode_table [31] = 'S';
```

write to enter this value. Also, find the scan code for the Enter key; then use '\ n '.

Adjust your previous code so that not only the scan code but also the character is output (if it is known, i.e. not 0 in the table). Test the program. (Of the printf- Format code for characters is% c. When outputting "Release" -Scan-
If you receive codes with negative numbers, first cast the scan code into one for output
int, to fix the problem.)

You now have a simple polling keyboard driver.

b) Interrupts: Now switch your driver to the use of interrupts.

(i) Add in the initialization of the system at a suitable place (e.g. in < *kernel main: initialize system>)* the commands

```
install_interrupt_handler (IRQ_KBD, keyboard_handler);
enable_interrupt (IRQ_KBD);
asm ("sti");                                    // enable interrupts
```

a (where IRQ_KBD already with # define on 1 is set: This is the interrupt number generated by the keyboard). You now have to implement the keyboard handler: it has the signature

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

and the system will automatically call it up whenever you press or release a key.

(ii) First check that the handler is called at all by only outputting a single character (e.g. '*') as confirmation and then entering the handler with return; ver
to let.

(iii) In the next step you output the values transmitted by the keyboard controller. For this you have to query the data port of the controller (as above); a test with the status port is not necessary, because the interrupt handler is only called when a key is pressed or released. After the output (as above also with the use of the scan code table), exit the handler with return;. Again, don't mind the problem of

scrolling screen, you need again clrscr () - Views when you reach the bottom.

(iv) The advantage of processing through interrupts is that you are in the main program (in Main)
take care of other things. Now create a function

```
void kreadline (char * s, int len);
```

which you from Main() out with (e.g.)

```
char input [41];                   // 40 characters plus \ 0 terminator
kreadline ((char *) input, 40);
```

call. The goal is that kreadline () the passed string (pointer to char) with the one
given characters (as far as they were recognized by your scan code table) until you either end the input with [RETURN] or until the maximum number len the signs
is enough; only then does the function return. The main program then outputs the read string; the whole thing repeats itself in an endless loop.

The trick here is to work with the interrupt handler. To do this, you need two new global variables that represent an input buffer and the next write position in the buffer:

```
char buffer [256];                    // buffer for inputs
short int pos = 0;                    // current position in the buffer
```

The interrupt handler should now work as follows:

- If the scan code is larger than 127 (release), it returns immediately return back.
- If an unknown scan code appears, it returns immediately return back.
- It outputs the letter on the terminal and also enters it in the buffer.
- Then he increases pos and returns return back.

kreadline () checks in an infinite loop whether (pos> 0 && buffer [pos-1] == '\ n') applies -
If this is the case, the function copies the entered string (from position 0 to pos-2)
to s, puts pos = 0 and returns. Note that the string must be null-terminated in order to be later used by printf () can be processed. You can use the '\ n'- Replace character with 0.

To copy the string, you can strncpy () use; the function works like the Linux function of the same name (man strncpy), expects target, source and maximum length of the string to be copied as arguments.

In order for everything to work, your scan code table must have an entry for the Enter key (in the right place must have '\ n ' stand.)

c) Bonus task: Expand the code so that you can use the backspace key to remove characters from the
delete the current entry. These should then also disappear from the screen (and in the end not via kreadline () land in the string).

10. Faults

The current version of the mini-kernel (from exercise 9) also contains fault handlers. Try these out for some typical faults:

a) Division by 0: Perform a division by 0 in the main program, e.g. B. with int z
= 1/0; - even if that generates a compiler warning.

b) Access unreachable storage, e.g. B. with

```
char * address = (char *) 0xE0000000; char tmp = * address;
```

c) Set an invalid segment in the segment register DS:

```
asm ("mov $ 32,% ax; mov% ax,% ds");
```

In return for your efforts, you should receive a Division by Zero Fault, a Page Fault and a General Protection Fault.

11. System Call Handler

In the folder tutorial05 / a version of the Ulix kernel can be found in your home directory, in
which the new code for System Call Handler has been implemented. It lies as a Literate Program (ulix.nw) which includes the sample solution for the keyboard interrupt handler.

In this task you develop a specific system call and try it out. When programming, make sure you have a *Literate Program* generate, i.e. incorporate code and documentation well into the overall document.

printf: The function printf () is available within the kernel, but processes would be
unable to call it. That is why in the first subtask you develop a syscall handler, the printf makes available for processes. To simplify matters, a function will be added later
userprint () that accepts exactly one string as an argument. (So we forego the option of passing a format string and any number of arguments.)

(i) First define in < *constants>* a syscall number for the printf- Syscall, e.g. B .:

```
# define __NR_printf                    1
```

(ii) Now write a syscall handler with the signature

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

of the function printf () calls. Make sure that you transfer the correct arguments - in which of the registers (accessible via r-> eax, r-> ebx, r-> ecx and r-> edx) can you find the address of the string?

(iii) Enter the new syscall handler in the syscall table. (iv) Write a function

```
void userprint (char * s);
```

which takes a string as an argument and then using one of the four syscall * () - Functions the system call carries out.

(v) Test the correct function by calling the main program

```
userprint ("test output \ n");
```

record, tape.