Vorbereitung

Im Rahmen dieser Übung werden Sie C-Code anschauen und auch geringfügig verändern. Wenn Sie unsicher sind, was Ihre Programmierkenntnisse angeht, suchen Sie sich bitte eine/n Partner/in für diese Übung.

Starten Sie die virtuelle Maschine und melden Sie sich diesmal als Benutzer `ulix` (nicht `fom`) mit dem Passwort `ulix` an. Starten Sie dann mit `startx` die grafische Oberfläche und öffnen Sie ein Terminalfenster.

Im Terminalfenster geben Sie den Befehl `update-ulix.sh` ein, das Kommando lädt einige Dateien vom Webserver (fom.hgesser.de) herunter.

1. Literate Programming – erste Schritte

Wechseln Sie in den Ordner `tutorial01/` – in dieser Aufgabe geht es darum, dass Sie sich ein wenig mit der Literate-Programming-Quelldatei vertraut machen.

a) Öffnen Sie zunächst die Datei `ulix.pdf` – sie enthält die PDF-Version des Literate Program `ulix.nw`. Lesen Sie darin Kapitel 1 (*Kernel-Layout*; nur eine halbe Seite) und Kapitel 7 (*Interrupts*). Der Code sollte Ihnen bekannt vorkommen: Er entspricht i. W. dem Code, den wir in der Vorlesung besprochen haben und den Sie auch in der PDF-Datei `ulix-interrupt-kapitel-muc.pdf` gesehen haben; im Vergleich zu diesem Auszug aus dem Ulix-basierten Lehrbuch sind die Erläuterungen deutlich knapper, damit die Datei überschaubar bleibt.

b) Öffnen Sie im Editor `gedit` die Datei `ulix.nw`, also mit dem Kommando

```
gedit ulix.nw &
```

(Dieser Editor bietet gegenüber `nedit` den Vorteil, dass Syntax Highlighting für LaTeX automatisch aktiviert ist.) Die Datei enthält die Beschreibung einer frühen Version des Ulix-Kernels, in der noch viele Features fehlen; der Code für die Interrupt-Behandlung, den Sie in der Vorlesung gesehen haben, ist aber bereits vorhanden.

Blättern Sie durch die Datei und suchen Sie dabei nach der Stelle, an der die Funktion `install_interrupt_handler()` implementiert wird. Der Prototyp der Funktion und die eigentliche Implementierung landen in verschiedenen Code-Chunks – welche sind das?

c) Einige Zeilen oberhalb der Fundstelle für Aufgabe **b)** wird im Chunk <*global variables*> das Array `void *interrupt_handlers[16]` deklariert. Warum hat es 16 Einträge? Eine Handler-Funktion `handler()` hat die Signatur `void handler (context_t *r);` `void*` ist ein „neutraler“ Pointer, und jeder Array-Eintrag speichert die (Start-) Adresse einer Handler-Funktion.

d) Mit `make` bauen Sie den Kernel neu, mit `make run` starten Sie das Ulix-System (im PC-Emulator `qemu`), und mit `make pdf` erstellen Sie die PDF-Dokumentation neu – testen Sie alle drei Schritte.

2. Tastaturtreiber: Polling und Interrupts

In dieser Aufgabe entwickeln Sie – basierend auf dem Code in `tutorial1/ulix.nw` – einen Tastaturtreiber. Achten Sie beim Programmieren darauf, dass Sie ein *Literate Program* erzeugen, also Code und Dokumentation gut in das Gesamtdokument einbauen.

a) **Polling:** Sie finden am Ende der Datei `ulix.nw` einen Abschnitt „Tastatur-Treiber“, in dem Sie Ihren neuen Code (oder den größten Teil davon) platzieren können. Bauen Sie nach jeder Teilaufgabe den Ulix-Kernel neu (`make`) und testen Sie ihn (`make run`).

Im ersten Schritt testen Sie, wie die Tastaturabfrage mit *Polling* funktioniert. Dazu benötigen Sie ein paar Informationen:

(i) Der Keyboard-Controller besitzt zwei Ports (`0x60` und `0x64`), die Sie mit `inportb()` auslesen können. Definieren Sie im Chunk <*constants*> die Portnummern:

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

Am Data-Port liegen Informationen über gedrückte / los gelassene Tasten an, über den Status Port können Sie prüfen, ob überhaupt eine Taste gedrückt wurde.

(ii) Versuchen Sie zunächst, in einer Schleife immer wieder den Data Port auszulesen und das Ergebnis (als Zahl) auszugeben: Sie können die Abfrage mit

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

erledigen. (Für Ihren Code ist ein leerer Code Chunk <*kernel main: user-defined tests*> am Ende der Noweb-Datei vorbereitet, der nach der Initialisierung aufgerufen wird.) Geben Sie den Scancode dann mit `printf()` aus. Das wird selbst bei einem `printf()`-Aufruf ohne `"\n"` relativ schnell den Bildschirm füllen. Darum bauen Sie vor den `printf()`-Aufruf eine Prüfung ein:

```
if (posy == 25) clrscr ();
```

Das Kommando löscht, wenn Sie am unteren Bildschirmrand angekommen sind, den Bildschirm, und die Ausgabe geht oben weiter. (`posy` enthält die aktuelle Zeile auf dem 80x25-Bildschirm.)

Sie werden feststellen, dass bei dieser Umsetzung ein ständiger (und schneller) Strom von Daten ausgegeben wird. Drücken Sie, während das System läuft, verschiedene Tasten; Sie sehen dann, dass sich die Ausgabe ändert. (Wahrscheinlich müssen Sie die Taste gedrückt halten, um eine Änderung zu erkennen.) Was dort angezeigt wird, ist ein Scan Code der Tastatur; jeder Wert entspricht einem Drücken oder Loslassen. Es erscheint immer wieder derselbe Wert, bis Sie erneut eine Taste drücken oder diese los lassen.

(iii) Verbessern können Sie Ihren Code, indem Sie auch das Status-Register über den Status-Port abfragen; das geht wie beim Daten-Port, nur eben mit `KBD_STATUS_PORT`. Wenn der zurückgegebene Wert das niedrigste Bit gesetzt hat (was Sie mit `if ((status & 1) == 1)` testen können), gibt es einen frischen Scan Code, und nur dann fragen Sie den Data-Port ab. In der geänderten Version erscheinen nur noch Ausgaben, wenn Sie eine Taste drücken oder los lassen.

(iv) Die Scan Codes für das Drücken und Loslassen einer Taste unterscheiden sich nur im gesetzten achten Bit; so sind etwa die Scan Codes für die Taste „A“ 30 und 158 (= 30 + 128), für das rechts davon liegende „S“ sind sie 31 und 159 (= 31 + 128). Erstellen Sie eine Zuordnungstabelle, die für einige Scan Codes den ASCII-Wert des jeweiligen Großbuchstabens enthält. Sie müssen dafür die ASCII-Werte nicht nachschlagen, sondern können z. B. `'A'` oder `'B'` in die Tabelle eintragen. Initialisieren Sie die Tabelle wie folgt mit Nullen:

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

Sie können dann z. B. für die Tasten „A“ und „S“, deren Scancodes (30, 158 bzw. 31, 159) Sie bereits kennen,

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

schreiben, um diesen Wert einzutragen. Suchen Sie auch den Scan Code für die Eingabetaste; als Zeichen verwenden Sie beim Eintragen dann `'\n'`.

Passen Sie Ihren bisherigen Code an, so dass nicht nur der Scan Code, sondern auch das Zeichen ausgegeben wird (falls es bekannt ist, also in der Tabelle nicht 0 steht). Testen Sie das Programm. (Der `printf`-Formatcode für Zeichen ist `%c`. Bei der Ausgabe von „Loslassen“-Scan-Codes erhalten Sie negative Zahlen, casten Sie für die Ausgabe den Scan Code zunächst in einen `int`, um das Problem abzustellen.)

Sie haben nun einen einfachen, pollenden Tastaturtreiber.

b) Interrupts: Jetzt stellen Sie Ihren Treiber auf die Verwendung von *Interrupts* um.

(i) Fügen Sie in die Initialisierung des Systems an geeigneter Stelle (z. B. in *<kernel main: initialize system>*) die Befehle

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

ein (wobei `IRQ_KBD` bereits mit `#define` auf `1` gesetzt ist: Das ist die Interrupt-Nummer, die die Tastatur erzeugt). Den Keyboard-Handler müssen Sie nun implementieren: Er hat die Signatur

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

und das System wird ihn automatisch immer dann aufrufen, wenn Sie eine Taste drücken oder sie loslassen.

(ii) Prüfen Sie zunächst, dass der Handler überhaupt aufgerufen wird, indem Sie darin nur zur Bestätigung ein einzelnes Zeichen (z. B. `'*'`) ausgeben und den Handler dann mit `return;` verlassen.

(iii) Im nächsten Schritt geben Sie die vom Keyboard-Controller übermittelten Werte aus. Dazu müssen Sie (wie oben) den Data-Port des Controllers abfragen; ein Test mit dem Status-Port ist nicht nötig, denn der Interrupt-Handler wird ja nur aufgerufen, wenn eine Taste gedrückt oder los gelassen wurde. Nach der Ausgabe (wie oben auch mit Einsatz der Scan-Code-Tabelle) verlassen Sie den Handler mit `return;`. Beachten Sie auch hier wieder das Problem des nicht scrollenden Bildschirms, Sie brauchen wieder `clrscr()`-Aufrufe, wenn Sie den unteren Rand erreichen.

(iv) Der Vorteil der Abarbeitung durch Interrupts ist, dass Sie sich im Hauptprogramm (in `main`) um andere Dinge kümmern können. Erstellen Sie nun eine Funktion

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

die Sie aus `main()` heraus mit (z. B.)

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

aufrufen. Das Ziel ist, dass `kreadline()` den übergebenen String (Pointer to `char`) mit den eingegebenen Zeichen (soweit sie von Ihrer Scan-Code-Tabelle erkannt wurden) füllt, bis Sie entweder mit [RETURN] die Eingabe beenden oder bis die maximale Zahl `len` der Zeichen erreicht ist; erst dann kehrt die Funktion zurück. Das Hauptprogramm gibt den gelesenen String denn aus; das Ganze wiederholt sich in einer Endlosschleife.

Der Trick besteht hier in der Zusammenarbeit mit dem Interrupt-Handler. Sie benötigen dazu zwei neue globale Variablen, die einen Eingabe-Puffer und die nächste Schreibposition im Puffer darstellen:

```
char buffer[256];       // Puffer fuer Eingaben
short int pos = 0;      // aktuelle Position im Puffe r
```

Der Interrupt-Handler soll nun wie folgt arbeiten:

– Ist der Scan Code größer als 127 (Loslassen), kehrt er sofort mit `return` zurück.

– Wenn ein unbekannter Scan Code auftaucht, kehrt er auch sofort mit `return` zurück.

– Er gibt den Buchstaben auf dem Terminal aus und trägt ihn auch in den Puffer ein.

– Danach erhöht er `pos` und kehrt mit `return` zurück.

`kreadline()` prüft in einer Endlosschleife, ob `(pos>0 && buffer[pos-1]=='\n')` gilt – wenn das der Fall ist, kopiert die Funktion den eingegebenen String (von Position 0 bis `pos-2`) nach `s`, setzt `pos=0` und kehrt zurück. Beachten Sie, dass der String Null-terminiert sein muss, damit er später von `printf()` verarbeitet werden kann. Sie können das `'\n'`-Zeichen durch 0 ersetzen.

Zum Kopieren des Strings können Sie `strncpy()` verwenden; die Funktion arbeitet wie die gleichnamige Linux-Funktion (`man strncpy`), erwartet also Ziel, Quelle und maximale Länge des zu kopierenden Strings als Argumente.

Damit alles funktionieren kann, muss Ihre Scan-Code-Tabelle einen Eintrag für die Eingabetaste besitzen (an der richtigen Stelle muss `'\n'` stehen.)

3. Faults

Die Version des Mini-Kernels (aus Aufgabe 2) enthält auch Fault-Handler. Probieren Sie diese für einige typische Faults aus:

a) Division durch 0: Führen Sie im Hauptprogramm eine Division durch 0 durch, z. B. mit `int z = 1 / 0;` – auch wenn das eine Compiler-Warnung erzeugt.

b) Greifen Sie auf nicht erreichbaren Speicher zu, z. B. mit

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

Als Lohn für Ihre Mühen sollten Sie einen *Division by Zero Fault* und einen *Page Fault* erhalten.

Teil 2

4. System Call Handler

Im Ordner `tutorial02/` in Ihrem Home-Verzeichnis finden Sie eine Version des Ulix-Kernels, in welche zusätzlich der Code für System Call Handler eingebaut wurde. Sie liegt als Literate Program (`ulix.nw`) vor, das u. a. die Musterlösung zum Tastatur-Interrupt-Handler enthält.

In dieser Aufgabe entwickeln Sie einen konkreten System Call und probieren diesen aus. Achten Sie beim Programmieren darauf, dass Sie ein *Literate Program* erzeugen, also Code und Dokumentation gut in das Gesamtdokument einbauen.

`printf`: Die Funktion `printf()` ist zwar innerhalb des Kernels verfügbar, Prozesse wären aber nicht in der Lage, sie aufzurufen. Darum entwickeln Sie in der ersten Teilaufgabe einen Syscall-Handler, der `printf` für Prozesse verfügbar macht. Zur Vereinfachung soll später eine Funktion `userprint()` bereitstehen, die genau einen String als Argument akzeptiert.

(i) Definieren Sie zunächst in <*constants*> eine Syscall-Nummer für den `printf`-Syscall, z. B.:

```
#define __NR_printf   1
```

(ii) Nun schreiben Sie einen Syscall-Handler mit der Signatur

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

der die Funktion `printf()` aufruft. Achten Sie darauf, dass Sie die richtigen Argumente übergeben – in welchem der Register (erreichbar über `r->eax`, `r->ebx`, `r->ecx` und `r->edx`) finden Sie die Adresse des Strings?

(iii) Tragen Sie den neuen Syscall-Handler in die Syscall-Tabelle ein.

(iv) Schreiben Sie eine Funktion `void userprint (char *s)`, die einen String als Argument nimmt und dann mit Hilfe einer der vier `syscall*()`-Funktionen den System Call durchführt.

(v) Testen Sie die korrekte Funktion, indem Sie ins Hauptprogramm den Aufruf

```
userprint ("Testausgabe\n");
```

aufnehmen.