Zum Auftakt booten (oder reaktivieren) Sie die Ulix-Devel-VM und führen in der Shell den Befehl `update-ulix.sh` aus. Damit laden Sie die Dateien herunter, die Sie für das Bearbeiten der aktuellen Übungsaufgaben benötigen.

5. Segmentierung

Im Ordner `tutorial01/` in Ihrem Home-Verzeichnis finden Sie eine frühe Version des Ulix-Kernels, die nur die Segmentdeskriptoren initialisiert.

a) Lesen Sie die Sourcecode-Dateien `ulix.c` und `start.asm`. Übersetzen Sie dann den Quelltext mit `make` und starten Sie den Kernel mit `make run`. Sie sollten die folgende Ausgabe erhalten:

```
  Booting 'ULIX-i386  (c) 2008-2011 Felix Freiling & Hans-Georg Esser'

root (fd0)
 Filesystem type is fat, using whole disk
kernel /ulix.bin
   [Multiboot-elf, <0x100000:0x46:0x0>, <0x100050:0xfb0:0x0>, <0x101000:0x0:0x9
000>, shtab=0x10a190, entry=0x10002a]


-



Hello World! This is not Ulix yet :)

address of main() [ulix.c]:    c01005b0
address of start  [start.asm]: 0010002a
stack: c0101008 - c0109008
```

b) Offensichtlich führt der Kernel die C-Funktion `main()` aus. Wie gelangt das System vom Assembler-Code (ab Label `start`) in die C-Funktion?

c) Die Datei `ulix.dump` enthält ein Listing des erzeugten Assembler-Codes. Sie finden hier u. a. alle Labels aus der Assembler-Datei `start.asm`, aber auch die Funktionsnamen aus der Datei `ulix.c`. Suchen Sie in der Datei die Labels `start`, `higherhalf` und `main` und schauen Sie, an welche Speicheradressen der zugehörige Code gelinkt wurde. (Die Speicheradressen stehen jeweils ganz links in Hexadezimalschreibweise ohne führendes „`0x`“.) Können Sie im Assembler-Code die Aktivierung der Trick-GDT erkennen? Er wird durch einen „long jump“ (`jmp`) ausgelöst, bei dem als Sprungadresse eine logische Adresse der Form `Segment:Adresse` angegeben ist. Suchen Sie außerdem die Labels `stack_first_address` und `stack_last_address` und vergleichen Sie die hierfür angezeigten Adressen mit der Ausgabe in der VM (letzte Zeile); sie sollten übereinstimmen.

d) Der Kernel verwendet die Funktion `printf()`, um Text auszugeben. Diese Funktion ist in der Datei `printf.c` implementiert, der dortige Code verwendet aber letzten Endes die Funktion `kputch()` (kernel put character), welche wieder in `ulix.c` definiert ist. Die `printf()`-Implementierung beschäftigt sich nur mit dem Verarbeiten der `printf`-typischen Formatparameter, etwa `%s` für Strings oder `%d` für Zahlen. Uns interessiert die Funktionsweise von `kputch()`.

Versuchen Sie zu verstehen, wie die Funktion `kputch()` Zeichen auf den Bildschirm schreiben kann. Googeln Sie ggf. nach „0xb8000 video“, um Informationen zu finden. Als Lösungshinweis eine kurze Erklärung zur Verwendung von Pointern: Mit den Befehlen

```
char *mem;  mem = (char*) 0x1234; *mem = 'a';
```

können Sie das Byte `'a'` (ASCII-Wert: `0x61`) in die Adresse `0x1234` des Speichers schreiben.

e) Warum verwendet in `kputch()` die folgende Zeile

```
screen = (char*) 0xc0000000 + 0xb8000 + posy*160 + posx*2;
```

die Multiplikatoren 160 und 2, und warum wird `0xc0000000` addiert?

f) Prüfen Sie mit dem Befehl `objdump -h ulix.bin`, welche Speicherbereiche die drei Sections `.setup`, `.text` und `.bss` verwenden. (Die weiteren Sections `.comment`, `.stab` und `.stabstr` können Sie ignorieren.) Vergleichen Sie die Werte mit den Angaben, die der Boot-Manager GRUB beim Laden des Kernels in der Zeile `[Multiboot-elf, …]` ausgibt.

6. Paging

Der Ordner `tutorial02/` in Ihrem Home-Verzeichnis enthält die nächste Variante des Ulix-Kernels: diesmal mit Paging.

a) Lesen Sie die Sourcecode-Dateien `ulix.c` und `start.asm` und lokalisieren Sie die im Foliensatz 5 vorgestellten Code-Ausschnitte. (`ulix.c` enthält noch zusätzlichen Code, den Sie in der Vorlesung nicht gesehen haben.)

b) Übersetzen Sie den Code mit `make` und starten Sie das Mini-Ulix mit `make run`.

c) In `ulix.c` hat sich die Funktion `kputch()` ein wenig verändert. Hier gibt es jetzt den folgenden Code-Block:

```
if (paging_ready)
    screen = (char*) 0xb8000 + posy*160 + posx*2;
else
    screen = (char*) 0xc0000000 + 0xb8000 + posy*160 + posx*2;
```

Hier wird die Variable `paging_ready` ausgewertet, die zunächst `false` ist und nach der Initialisierung des Pagings auf `true` gesetzt wird. In dem Fall fällt bei der Adressberechnung die Addition von `0xc0000000` weg (vgl. Aufgabe 5d). Warum funktioniert das?

7. Literate Programming

a) Konvertieren Sie die beiden Dateien `ulix.c` und `start.asm` (aus `tutorial02/`) in ein Literate Program namens `tutorial02.nw`. Die Dokumentation, die Sie ergänzen, kann aus Stichworten bestehen, und Sie können sich an den Folieninhalten orientieren.

b) Testen Sie, dass Sie aus dem Literate Program wieder die ursprünglichen (oder ähnliche, ebenfalls erfolgreich kompilierbare) Code-Dateien extrahieren können.

c) Erzeugen Sie auch eine LaTeX-Datei und daraus eine PDF-Datei. Schicken Sie mir die `nw`-Datei (das Literate Program) und die `pdf`-Datei per E-Mail zu (→ h.g.esser@cs.fau.de), ich gebe Ihnen dann ein Feedback zur Umsetzung. (Dieser Teil ist freiwillig, aber empfohlen.)

Hausaufgabe: Lektüre

Laden Sie von der Webseite über den Link Kapitel 10+11 die PDF-Datei *ulix-book-chap10-11.pdf* herunter. Sie enthält eine ausführlichere Beschreibung von einigen der in Foliensatz 5 behandelten Themen (Booten, Segmentierung, Struktur der Kernel-Quellen). Lesen Sie den Text – gibt es hier Stellen, die ausführlicher behandelt werden sollten oder die unklar sind? Dann schicken Sie mir bitte eine Mail (→ h.g.esser@cs.fau.de).