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.

12. User-Mode-Programme

Im Ordner `tutorial06/` in Ihrem Home-Verzeichnis finden Sie eine Version des Ulix-Kernels, in welche der neue Code für das Umschalten auf den User Mode eingebaut wurde. Sie liegt als Literate Program (`ulix.nw`) vor, das u. a. die Musterlösung zum Tastatur-Interrupt-Handler enthält.

In dieser Aufgabe bringen Sie eine erste User-Mode-Anwendung zum Laufen.

Wie üblich: Achten Sie beim Programmieren darauf, dass Sie ein *Literate Program* erzeugen, also Code und Dokumentation gut in das Gesamtdokument einbauen.

a) **Testprogramm:** Zunächst erstellen Sie ein kleines Testprogramm für ULIX, damit Sie sehen, wo die Reise hin geht. Das eigentliche Hauptprogramm `main()` in der Datei `test.c` soll nur aus folgenden Zeilen bestehen:

```
int main () {
  printf ("Hallo - User Mode!\n");
  for (;;) ;   // Endlosschleife
}
```

Die Funktion `printf()` müssen Sie noch selbst definieren: Binden Sie dazu eine der `syscall*`-Funktionen (`syscall1`, `syscall2` etc.) in die Datei ein. Unsere simple `printf`-Variante akzeptiert nur ein einziges String-Argument, arbeitet also wie die Funktion `userprint()` aus der letzten Übung. Die Konstante `__NR_printf` (`1`) müssen Sie auch mit `#define` definieren.

Wichtig: In unseren User-Mode-Programmen muss die `main`-Funktion immer ganz am Anfang stehen. Funktionen, die Sie aus `main()` heraus aufrufen (also z. B. `printf`) müssen Sie *oberhalb* von `main()` vorausdeklarieren (Prototyp), aber *unterhalb* von `main()` implementieren! Das liegt daran, dass ULIX später das erzeugte Programm an die virtuelle Adresse 0 (und folgende) lädt und die Ausführung auch an der Adresse 0 beginnt – der Code des Hauptprogramms (`main`) muss also ganz oben in der erzeugten Programmdatei stehen.

Für das Übersetzen mit dem Compiler ist bereits der nötige `gcc`-Aufruf in das Makefile eingetragen, Sie müssen daran nichts ändern:

```
   $(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
```

Das erzeugt (wenn Sie `make` eingeben) aus dem Quellcode in `test.c` das ausführbare Programm `test`.

b) **Disassembler installieren:** Sie benötigen für diese Aufgabe einen Disassembler, der aus einer binären Programmdatei wieder lesbaren Assembler-Quelltext macht. Falls Sie bei der Eingabe von `x86dis` eine Fehlermeldung („Kommando nicht gefunden“) erhalten, installieren Sie den Disassembler mit dem folgenden Kommando nach:

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

c) Disassemblieren Sie das erzeugte Programm `test` mit dem Befehl

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

Die Ausgabe sollte wie folgt anfangen:

```
00000000 8D 4C 24 04                    lea   ecx, [esp+0x4]
00000004 83 E4 F0                       and   esp, 0xF0
00000007 FF 71 FC                       push  [ecx-0x4]
0000000A 51                             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
...
```

Das ist der Anfang der übersetzten `main()`-Funktion; der `call`-Befehl ruft die `printf()`-Funktion auf.

d) Da wir kein Dateisystem haben, auf das wir das Programm kopieren könnten, nutzen wir einen Trick: Wir schreiben den Code direkt in den Kernel und kopieren ihn später in den User-Mode-Speicher. Hierbei hilft das Tool `hexdump`, dessen Ausgabeformat sich über einen Formatstring einstellen lässt:

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

Das erzeugt eine Ausgabe der folgenden 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,
*
```

Vergleichen Sie diese Hex-Zahlen mit den vom Disassembler ausgegebenen Hex-Zahlen: Sie sind identisch.

Diese Ausgabe können Sie (ohne die Zeile mit dem Sternchen) in den ULIX-Quellcode (in <*global variables*>) übernehmen. Deklarieren Sie in **Kapitel 12.6** `usermodeprog` in der Form

```
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,
  [...]
};
```

Dadurch landen die einzelnen Bytes in einem passenden Array. Wenn es am Ende eine Zeile gibt, die nur Nullen (`0x00`) enthält, können Sie diese auch entfernen.

Damit können Sie nun eine Funktion schreiben, die das Programm lädt.

e) Fügen Sie im ULIX-Code im Chunk <*kernel main: user-defined tests*> am Ende die Zeile

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

ein: Die Funktion `start_program_from_ram` (die Sie noch teilweise implementieren müssen) wird das Programm laden und es starten, wozu es vom Kernel-Mode (Ring 0) in den User-Mode (Ring 3) wechselt.

13. User Mode aktivieren

In dieser Aufgabe implementieren Sie in ULIX einige der nötigen Methoden, um ein Programm zu starten. Viele Codeteile aus der Vorlesung sind bereits integriert.

a) Die wichtigste Funktion ist `start_program_from_ram()`: Sie funktioniert ähnlich wie die in der Vorlesung vorgestellte Funktion `start_program_from_disk()`, die wir nicht verwenden können – Unterschiede sind: Laden aus dem Speicher (statt von Platte), keine Berücksichtigung des Schedulers (wir starten zunächst nur einen einzigen Prozess).

Sie müssen den folgenden Code nicht abtippen; er ist bereits in `ulix.nw` enthalten.

```
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
  cpu_usermode (BINARY_LOAD_ADDRESS,
                TOP_OF_USER_MODE_STACK);    // jump to user mode
};
```

b) Um die fehlenden Chunks zu implementieren, sind folgende Schritte notwendig:

Füllen Sie (in *<start program from ram: prepare address space and TCB entry>*, **Kapitel 12.4**) die beiden Variablen `as` und `tid` mit Inhalten, indem Sie die Funktionen `create_new_address_space()` und `register_new_tcb()` aufrufen. Diese Funktionen arbeiten wie in der Vorlesung gezeigt und sind in der `ulix.nw`-Datei der heutigen Übung bereits enthalten. Beachten Sie dabei nur die Aufrufreihenfolge: `register_new_tcb()` benötigt die Address Space ID als Argument. Geben Sie neuen Prozessen 64 KByte Programmspeicher und 4 KByte User Mode Stack.

Weitere Aufgaben für den ersten Code Chunk sind, die Thread-Control-Block-Elemente mit sinnvollen Werten zu füllen. Als PPID (Parent Process ID) vergeben Sie für den neuen (ersten) Prozess den Wert 0.

Der Code-Chunk *<start program from ram: create kernel stack>* ist weitgehend mit dem in der Vorlesung gezeigten Code Chunk *<start program from disk: create kernel stack>* identisch; er ist im Programm bereits enthalten.

Es fehlt noch der Code-Chunk *<start program from ram: load binary>*: In diesem müssen Sie mit `memcpy()` den Programmcode aus dem Array, dessen Adresse und Länge Sie der Funktion in den Parametern `address` und `size` übergeben haben, an die Adresse 0 kopieren.

`cpu_usermode()` ist eine Assembler-Funktion; Sie finden diese in `start.asm`.

c) Testen Sie Ihren Code (mit `make` und `make run`) – ULIX sollte nach der allgemeinen Begrüßung („Hallo Welt“) auch die Zeile aus dem User-Mode-Programm („Hallo - User Mode!“) ausgeben.

14. Mehr Features für den User Mode

In dieser Aufgabe erweitern Sie die Features des User Mode. Ziel ist, dass das Hauptprogramm im User-Mode-Programm mit `readline()` Text von der Tastatur einlesen kann. Dazu sind – in **Kapitel 12.7** – mehrere Schritte nötig:

a) Schreiben Sie einen System-Call-Handler, der die Funktion `kreadline()` aufruft. Wie alle Syscall-Handler hat er die Signatur

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

Wenn er aufgerufen wird, enthält `r->eax` die Syscall-Nummer (die Sie ignorieren können), und `r->ebx` soll die Adresse eines im User-Mode-Programm deklarierten Strings enthalten. Reservieren Sie dafür im User-Mode-Programm (`test.c`) eine Variable, die 256 Zeichen aufnehmen kann. Die Länge übergeben wir nicht, Sie können diese beim Aufruf von `kreadline()` auf 256 setzen. Im Handler müssen Sie vor dem Aufruf von `kreadline()` die Interrupts aktivieren, denn durch den Sprung in den Handler werden diese deaktiviert. Fügen Sie darum diese Zeile ein:

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

Vergeben Sie eine Syscall-Nummer `__NR_readline` und tragen Sie den Handler in die Syscall-Tabelle ein.

b) Schreiben Sie jetzt in `test.c` eine Funktion `void readline (char *s);` welche über eine der `syscall*`-Funktionen den richtigen System Call durchführt; dazu müssen Sie auch in der Datei `test.c` die Konstante `__NR_readline` definieren (vgl. Aufgabe 12a). Es gilt auch hier wieder: Die neue Funktion müssen Sie oberhalb von `main()` vorausdeklarieren und unterhalb von `main()` implementieren.

c) Passen Sie die Funktion `main()` in `test.c` so an, dass sie in einer Endlosschleife Text einliest und wieder ausgibt, also z. B.

```
int main () {
  char s[256];
  printf ("Hallo - User Mode!\n");
  for (;;) {
    printf ("> ");  readline (s);
    printf ("Eingabe war: ");  printf (s);
  }
}
```

d) Testen Sie Ihr Programm. `test.c` müssen Sie nach dem Kompilieren mit `make` wieder wie in Aufgabe 12 d) bis e) in Hex-Code umwandeln und in den Kernel-Code in `ulix.nw` integrieren. Danach rufen Sie `make` erneut auf, um den angepassten Kernel zu übersetzen.

Die Ausgabefunktion beherrscht mittlerweile Scrolling, so dass Sie auch weiter testen können, wenn Sie am unteren Bildschirmrand ankommen. Schauen Sie sich zum Verständnis des Scrolling-Features die Funktion `scroll()` und die beiden Stellen, an denen sie aufgerufen wird, an.

e) Die Funktion `scroll()` bestimmt (ebenso wie `kputch()`) die richtige Speicheradresse u. a. durch Auswerten der Variable `VIDEO`: Suchen Sie die Stellen im Code, an denen `VIDEO` verändert wird – die Variable nimmt während der Initialisierung drei verschiedene Werte an (`0xc00b8000`, `0xb8000` und `0xd00b8000`): Woran liegt das? Zur Erinnerung: Die physischen Speicheradressen, an denen Sie den Text-Framebuffer der Grafikkarte finden, beginnen bei `0xb8000`.

f) Zusatzaufgabe: Übernehmen Sie den Code in `test.c` in das Literate Program `ulix.nw` und passen Sie das Makefile so an, dass auch die Datei `test.c` aus diesem Literate Programm extrahiert wird. Als Code-Chunk-Namen für den Code in `test.c` verwenden Sie dazu *<test.c>*. Sie brauchen dann einen zusätzlichen Aufruf von `notangle` analog zu

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

wobei Sie in der Option `-R` den neuen Chunknamen verwenden und auch die Ausgabeumleitung mit `>` anpassen. Beachten Sie, dass im Makefile alle Befehle mit [Tab] eingerückt sein müssen.

Fassen Sie Code, der sowohl in `ulix.c` als auch `test.c` identisch vorkommt, durch geeignete Code Chunks zusammen, die Sie in beiden Dateien einbauen – dann stehen die bisher doppelten Code-Zeilen im Literate Program nur noch je einmal.