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.

16. Musterlösung Übungsblatt 7

Im Ordner `loesung06/` in Ihrem Home-Verzeichnis finden Sie die Musterlösung zur letzten Übung. Starten Sie die in diesem Ordner liegende Ulix-Version mit `make` und `make run`, und überzeugen Sie sich durch einen Blick in `ulix.nw` und `test.c` davon, dass die Schleife, die eine Zeile Text einliest und wieder ausgibt, im Usermode läuft. In der Datei `ulix.pdf` können Sie auch die letzten Kapitel mit der lesbar dokumentierten Fassung der Musterlösung durchlesen.

17. Scheduling mit der Esc-Taste

Im Ordner `tutorial07/` in Ihrem Home-Verzeichnis finden Sie eine Version des Ulix-Kernels, in welche die neuen Funktionen `fork()` und `schedule()` aus der Vorlesung (Foliensatz 10) eingebaut wurden. Sie liegt als Literate Program (`ulix.nw`) vor.

Der Scheduler wird in dieser Ulix-Version allerdings nie aufgerufen – normal wäre dafür der Timer-Handler zuständig, den es noch nicht gibt. In dieser Aufgabe richten Sie ein „manuelles Scheduling“ ein, das Sie durch Drücken der Taste [Esc] auslösen können.

a) Betrachten Sie zunächst das User-Mode-Programm `test.c`, dessen in Maschinencode übersetzte Version bereits (mit derselben Methode wie in der letzten Übung) in den Kernel integriert wurde. Seine `main()`-Funktion erzeugt mit `fork()` einen zweiten Prozess, Vater und Sohn geben danach in Endlosschleifen „V“ bzw. „S“ aus:

```
int main () {
  printf ("Hallo - User Mode!\n");
  int pid = fork ();
  for (;;) {
    if (pid == 0) printf ("S");  // Sohn
    else          printf ("V");  // Vater
  }
}
```

Starten Sie die Ulix-Version (mit `make` und `make run`) – wie Sie sehen, werden nur „V“s ausgegeben, es läuft also nur der Vaterprozess. (Wenn Sie unter Linux eine C-Datei erstellen, die nur diese `main()`-Funktion enthält, erscheinen nach dem Kompilieren und Starten des Programms die Buchstaben im Wechsel.)

b) Der Scheduler steckt in der Funktion `scheduler()`, wie in der Vorlesung vorgestellt. Für erste Tests bauen Sie nun in den Keyboard-Handler eine Abfrage ein, ob die Esc-Taste gedrückt wurde – sie hat den Scancode 1. Ergänzen Sie also in der Funktion `keyboard_handler()` hinter dem Auslesen des Scancodes `s` mit `inportb()` einen Test auf Scancode 1:

```
if (s == 1) {
  scheduler (r, 0);
  return;
}
```

Wenn Sie jetzt den Kernel neu kompilieren und starten, erscheinen wieder nur „V“s – durch Drücken von [Esc] können Sie aber auf den Sohnprozess umschalten, und es erscheinen nun

„S“-Buchstaben. Jedesmal, wenn Sie erneut [Esc] drücken, wechseln Sie zwischen den beiden Prozessen hin und her.

18. Scheduler-Aufruf im Timer-Handler

In dieser Aufgabe ergänzen Sie einen Timer-Handler, über den Ulix den Scheduler regelmäßig aufrufen und damit zwischen mehreren Prozessen hin und her wechseln kann.

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

a) Entfernen Sie zunächst den Aufruf des Schedulers aus dem Keyboard-Handler, den Sie in der vorherigen Aufgabe eingefügt haben – Sie können ihn auch einfach auskommentieren.

b) Wir wollen den Timer-Handler u. a. dazu nutzen, festzustellen, wie lange Ulix bereits läuft. Definieren Sie dazu eine globale Variable `unsigned long int ticks`, die Sie auf 0 initialisieren. (In welchen Code Chunk gehört diese Variable?) Tragen Sie den Code zu Aufgabe b) und c) in Kapitel 14 („Den Scheduler aktivieren“) von `ulix.nw` ein – es ist nicht nötig, an frühere Stellen im Dokument zu springen und dort etwas zu ergänzen.

Schreiben Sie dann eine Funktion `timer_handler`, welche dieselbe Signatur wie alle Interrupt-Handler hat, als Beispiel dient der Keyboard-Handler. Als erste Aufgabe soll diese Funktion die globale Variable `ticks` inkrementieren. Rufen Sie danach den Scheduler mit

```
scheduler (r, 0);
```

auf. Der Timer-Handler ist damit für unsere Zwecke schon funktionsfähig, aber Sie müssen ihn noch registrieren und den Interrupt einschalten. Ergänzen Sie darum im Code Chunk *<kernel main: initialize system>* geeignete Funktionsaufrufe – auch hierzu können Sie sich daran orientieren, wie Ulix den Keyboard-Handler einträgt. Sie benötigen für diesen Schritt die Interrupt-Nummer des Timer-Interrupts, die in der Konstanten `IRQ_TIMER` definiert ist (sie ist =0).

Übersetzen und starten Sie den so angepassten Kernel. Jetzt sollte die Ausgabe von „V“s und „S“s automatisch wechseln, und zwar relativ schnell: Jedesmal, wenn ein Timer-Interrupt ausgelöst wird, schaltet Ulix auf den jeweils anderen Prozess um.

c) Das Umschalten zwischen den Prozessen kostet Rechenzeit, darum ist es sinnvoll, etwas längere Pausen zwischen den Umschaltvorgängen einzurichten, einen aktiven Prozess also ein wenig länger rechnen zu lassen, bevor der nächste dran kommt.

Das können Sie erreichen, indem Sie im Timer-Handler nicht einfach `scheduler()` aufrufen, sondern dies nur tun, wenn `ticks` das Vielfache einer bestimmten Zahl ist. Mit

```
if (ticks % 5 == 0)
```

prüfen Sie z. B., ob `ticks` ein Vielfaches von 5 ist. Rufen Sie den Scheduler nur dann auf, wenn diese Bedingung erfüllt ist, springen Sie im Ergebnis nur bei jedem fünften Timer-Interrupt in den Scheduler. Passen Sie den Code entsprechend an und prüfen Sie, wie sich das Verhalten für unterschiedliche Werte (z. B. 5, 25, 100) ändert.

d) Der aktuelle Code in `scheduler()` unterstützt (anders als die in der Vorlesung besprochene Variante) nur exakt zwei Prozesse, deren Verwaltungsdaten in den TCB-Einträgen 1 und 2 stehen. Dadurch ist es viel leichter, den jeweils nächsten Prozess zu bestimmen (eben immer 1 und 2 im Wechsel). Wie müssten Sie die Funktion anpassen, um exakt vier Prozesse zu unterstützen?