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. System Call Handler

Im Ordner `tutorial05/` in Ihrem Home-Verzeichnis finden Sie eine Version des Ulix-Kernels, in welche der neue 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 einige konkrete System Calls und probieren diese aus. Achten Sie beim Programmieren darauf, dass Sie ein *Literate Program* erzeugen, also Code und Dokumentation gut in das Gesamtdokument einbauen.

a) **`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. (Wir verzichten also hier auf die Möglichkeit, einen Format-String und beliebig viele Argumente zu übergeben.)

(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 (struct regs *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.

b) **`kpeek`:** Sie wollen Prozessen die Möglichkeit einräumen, beliebige Speicherstellen (des Kernel-Speichers) auszulesen. Dazu brauchen Sie eine Funktion

```
int kpeek (unsigned int address);
```

die als Argument eine Adresse akzeptiert und das dort gespeicherte Byte (also einen Wert zwischen 0 und 255) zurück gibt. Falls die Adresse nicht verfügbar ist, soll die Funktion `-1` zurück geben.

Als reine Kernel-Funktion könnten Sie `kpeek()` wie folgt implementieren:

```
int kpeek (unsigned int address) {
  int page = address / 4096;
  if (pageno_to_frameno (page) == -1)
    return -1;
  else
    return  *(char*)address;
}
```

Diese Funktion wäre aber wieder nur für den Kernel erreichbar. Implementieren Sie stattdessen `kpeek()` über einen System Call. Dabei gehen Sie grundsätzlich vor wie in Aufgabe **a)**:
– Vergabe einer Syscall-Nummer,
– Entwickeln eines Syscall-Handlers (in dem dann eine Variation des obigen Codes auftaucht, die die Parameter- und Rückgabewert-Übergabe über Register berücksichtigt),
– Eintragen des Handlers in die Handler-Tabelle,
– Schreiben der Funktion `kpeek()`, die den richtigen System Call über eine der Funktionen `syscall1()` bis `syscall4()` aufruft

Testen Sie das korrekte Funktionieren mit folgenden Zeilen:

```
unsigned int address = 0xc0000000;
*(char*)(address) = 123;
printf ("Teste kpeek: %d\n", kpeek (address));
```

(Der mittlere Befehl schreibt direkt den Wert 123 an die Speicherstelle `address`.) Im Ergebnis soll das System „Teste kpeek: 123“ ausgeben. Probieren Sie auch den Aufruf mit einer Adresse, die nicht existiert, z. B. mit

```
printf ("Teste kpeek/fail: %d\n", kpeek (0x90000000));
```

(Hier sollte das System „Teste kpeek/fail: -1“ ausgeben.)

c) **`kpoke`:** Neben `kpeek()` wollen Sie auch `kpoke()` unterstützen: Damit können Prozesse beliebige Speicherstellen des Kernels verändern. Die Signatur der Funktion ist

```
void kpoke (unsigned int address, unsigned char value);
```

Wenn die Adresse verfügbar ist, soll der Wert `value` an die Speicherstelle geschrieben werden; falls nicht, soll die Funktion einfach zurück kehren. Auch hier ist die Implementierung direkt im Kernel einfach, der nötige Code wäre

```
void kpoke (unsigned int address, unsigned char value) {
  if ( ... )   // Test auf Verfügbarkeit
    *(char*)(address) = value;
  return;
}
```

Aber auch das geht wieder nicht (siehe oben). Implementieren Sie `kpoke()` über einen System Call und testen Sie die korrekte Funktion mit folgendem Code:

```
unsigned int address = 0xc0000000;
kpoke (address, 123);
printf ("Teste kpoke: %d\n", kpeek (address));
```

Das ist derselbe Test wie in Aufgabe **b)**, nur dass Sie hier mit `kpoke()` in den Speicher schreiben.