Prof. Dr. Th. Letschert
FB MNI 13. März 2002

Übung 1 - Betriebssysteme I

Aufgabe 0: Basiswissen

Rechnerarchitektur:

- Aus welchen Komponenten besteht ein Rechner mit Von-Neumann–Architektur?
- Was sind Bits und Bytes? Was ist ein Register, was ist ein Befehlszähler?
- Was versteht man unter RAM?
- Was ist ein Maschinenbefehl?
- Was ist eine Unterbrechung?
- Wann ist ein Rechner ein “32 Bit–Rechner”?
- Welche Komponente führt Befehle aus? Warum kann ein Befehl nicht direkt von Platte geladen werden?
- Was ist eine Unterbrechung, welche Varianten von Unterbrechungen gibt es?
- Was ist eine Software–Unterbrechung? Gibt es Unterbrechungen die keine Software–Unterbrechungen sind?
- Was ist ein “Interrupt–Vektor” und was ist eine “Interrupt–Service–Routine”?

Programmierung:

- Was versteht man unter einem “Assembler” und was ist “Assemblercode”?
- Erläutern Sie die Begriffe *Interpreter* (auch: *Interpretierer*), *Compiler*, *Binder* und *Lader*.
- Was versteht man unter getrennter Übersetzung? Erläutern Sie warum bereits Ihr allererstes C/C++–Programm – Hallo–Welt oder ähnliches – auf getrennter Übersetzung basierte.
- Beschreiben Sie möglichst genau, welche Informationen eine *Include*–Datei enthält und von wem und wozu diese Informationen benötigt werden. Nutzen Sie in Ihrer Erklärung folgendes Beispiel:

  ```
  #include <math.h>   // in C++: #include <cmath>
  int main () {
    double d = sqrt (10);
  }
  ```

  Was ist `math.h`? Welchen Bezug haben `math.h` und `sqrt`? Wo ist der Code der `sqrt`–Funktion in welcher Form zu finden?

Aufgabe 1

1. Was ist ein absolut und was ein relativ adressiertes Maschinenprogramm?
2. Warum können nur relativ adressierte Maschinenprogramme an eine vom Lader dynamisch zugewiesene Hauptspeicheradresse geladen werden?
3. Welche Vorteile bringen relativ adressierte Maschinenprogramme dem System und seinen Benutzern?
4. Was versteht man unter Segmentierung und welche Vorteile bringt es dem System und seinen Benutzern?
5. Was ist eine statische und was eine dynamische Bibliothek. Was sind die jeweiligen Vor– und Nachteile?

Aufgabe 2

1. Was ist ein Systemaufruf?
2. Wie und warum unterscheiden sich normale Funktionsaufrufe von Systemaufrufen?
3. Welche Unterstützung liefert die Hardware bei einem Systemaufruf?
4. Warum muss ein Programm mit einem Systemaufruf beendet werden?
5. Vergleichen Sie den Aufruf eines Unterprogramms (z.B. einer Funktion in C) mit dem Start eines Programms. Was passiert bei einem Funktionsaufruf, was bei einem Programmstart?
6. Übersetzen Sie die beiden folgenden Programme und vergleichen Sie die Größe der erzeugten ausführbaren Programme. Erklären Sie den Unterschied!

```
//Progr. 1 -------------------
char buf[50000] = "0";
int main () {
  return 0;
}
//----------------------------

//Progr. 2 -------------------
int main () {
  char buf[50000] = "0";
  return 0;
}
//----------------------------
```

7. In C++ sollte die Funktion `main` den Typ `int` haben. Sie kann einen Int–Wert zurück geben, muss aber nicht. Ein fehlendes `return` in `main` entspricht nach C++–Sprachstandard einem `return 0;`. Stellen Sie fest, ob der GNU–Compiler sich entsprechend verhält. Übersetzen Sie dazu Testprogramme mit der Option `-S` in Assemblercode.
8. Wohin kehrt die `main`–Funktion eines C/C++–Programms zurück? Ins Betriebssystem? Was genau ist der Unterschied zwischen `return 0;`, `exit(0);` und `_exit(0);`?

Aufgabe 3

Vergleichen Sie die beiden folgenden Programme:

```
Progr. 1 (C++): -------------------------
#include <iostream>
using namespace std;
int main () {
  cout << "Hallo\n";
}
----------------------------------------

Progr. 2 (C): ---------------------------
#include <stdio.h>
int main () {
  printf ("Hallo\n");
  return 0;
}
//--------------------------------------
```

1. Übersetzen und binden Sie die beiden Programme einmal statisch und einmal dynamisch. Lassen Sie sich von `truss` bzw. `strace` ausgeben, welche Systemaufrufe jeweils abgesetzt werden.

2. Finden heraus, mit welchem Systemaufruf die beiden Programme sich jeweils beim Betriebssystem melden, um den Text “Hallo” auszugeben.

3. In der Datei `hallo-sys.c` befinde sich folgendes Programm:

   ```
   #include <unistd.h>
   char buf[7] = "Hallo\n";
   int main () {
      write(1, buf, 6);
      return 0;
   }
   ```

   Was bedeutet der Aufruf `write(1, buf, 6);`? Welche Bedeutung haben die Parameter? Konsultieren Sie das Manual (die *“man–pages”*).

   Was ist der Unterschied zwischen diesem Programm und den beiden vorhergehenden?

Aufgabe 4

1. Betrachten Sie folgendes Assemblerprogramm (in GNU–Assembler):

   ```
   .globl buf
   .data
           .type    buf,@object
           .size    buf,7
   buf:
   ```

```
        .string "Hallo\n"

        .section .text
        .globl _start

_start:
        movl %esp,%ebp
        subl $8,%esp
        addl $-4,%esp

        ## Aufruf write () system call
        pushl $7                # Parameter 1: 7
        pushl $buf              # Parameter 2: Adresse von buf
        pushl $1                # Parameter 3: Deskriptor 1
        call write

        ## Programm-Ende via _exit () system call
        pushl $0                # Parameter 1: 0
        call _exit
```

Dieses Programm kann nicht aus einem C oder C++–Quellprogramm erzeugt worden sein. Warum nicht?

2. Wer oder was ruft die `main`–Funktion eines C/C++–Programms auf? Warum dieser Umstand, warum wird `main` nicht einfach als Einstiegspunkt des Programms übersetzt?

3. Betrachten Sie folgendes Assemblerprogramm:

```
.globl buf
.data
        .type    buf,@object
        .size    buf,7
buf:
        .string "Hallo\n"
        .section .text
        .globl _start

_start:
        ## Ausgabe von Hallo
        xorl %ebx, %ebx         # %ebx = 0
        movl $4, %eax           # eax = 4 = write
        xorl %ebx, %ebx         # %ebx = 0
        incl %ebx               # %ebx = 1 = stdout
        leal buf, %ecx          # %ecx <--- buf
        movl $7, %edx           # %edx = 7
        int $0x80               # Interrupt Systemaufruf

        ## Programm beenden
        xorl %eax, %eax         # %eax = 0
        incl %eax               # %eax = 1 = _exit
        xorl %ebx, %ebx         # %ebx = 0 return code
        int $0x80               # Interrupt Systemaufruf
```

Wodurch unterscheidet sich dieses Programm vom vorhergehenden?

4. Erläutern Sie den `int`–Befehl mit seinem Argument.

5. Wie muss ein Programm aussehen, das HALLO ausgibt, aber völlig ohne getrennte Übersetzung auskommt, ein Programm, also das keinen getrennt übersetzten Code in irgendwelchen Objektdateien oder Bibliotheken benutzt und (in übersetzter) Form direkt geladen und ausgeführt werden kann?

Aufgabe 5: Bonusaufgabe 1, Termin 7. April 2002

Bonusaufgaben gelten als termingerecht abgegeben, wenn sie in einem Praktikum *vor* dem angegeben Termin erfolgreich präsentiert wurden.

1. Betrachten Sie folgendes Programm in C++:

```
#include <iostream>
using namespace std;
int main () {
  cout << "Hallo\n";
}
```

   Übersetzen und binden Sie das Programm einmal statisch und einmal dynamisch. Stellen Sie die Größe der Objektdatei und des ausführbaren Codes jeweils in beiden Varianten fest. Erklären Sie den Unterschied.

2. Schreiben Sie in C++ ein äquivalentes Hallo–Programm. Ihr Programm soll nicht die Standardbibliothek benutzen, sondern direkt Systemaufrufe des Betriebssystems.

   Stellen Sie die Größe der Objektdatei und des ausführbaren Codes jeweils bei statischer und bei dynamischer Bindung fest. Vergleichen Sie dies mit der ersten Version.

3. Betrachten Sie folgendes Hallo–Programm in GNU–Assembler für Linux.

```
.globl buf
.data
        .type  buf,@object
        .size  buf,7
buf:
        .string       "Hallo\n"

        .section .text
        .globl _start

_start:
        ## Aufruf write () system call
        xorl %ebx, %ebx       # %ebx = 0
        movl $4, %eax         # eax = 4 = write ()
        xorl %ebx, %ebx       # %ebx = 0
        incl %ebx             # %ebx = 1 = stdout
        leal buf, %ecx        # %ecx ---> buf
```

```
        movl $7, %edx          # %edx = 7
        int $0x80              # ?????

        ## Programm-Ende via _exit () system call
        xorl %eax, %eax        # %eax = 0
        incl %eax              # %eax = 1, _exit () system call
        xorl %ebx, %ebx        # %ebx = 0 normal program return code
        int $0x80              # ?????
```

In wie weit ist dieses Programm Plattform–spezifisch? Welche Bedeutung hat der `int`–Befehl?

4. Betrachten Sie folgendes Hallo–Programm in Assembler für SPARC/Solaris Rechner:

```
        .global buf
.section        ".data"
        .align 8
        .type    buf,#object
        .size    buf,7
buf:
        .asciz  "Hallo\n"
.section        ".text"
        .globl _start
_start:
        mov     1, %o0
        sethi   %hi(buf), %o2
        or      %o2, %lo(buf), %o1
        mov     7, %o2
        call    write, 0
        nop
        call    exit, 0
        nop
```

(Die Anweisungen sind jeweils mit TAB eingerückt.)

Übersetzen und binden Sie es (statisch) mit der/den notwendigsten Bibliothek(en) (welchen Code muss/müssen diese bieten) und bringen es zur Ausführung.