Systemnahe Software I
WS 2018/2019

Andreas F. Borchert
Universität Ulm

18. Oktober 2018

Widmung

Ausschnitt eines Fotos von Denise Panyik-Dale, CC-BY-2.0

Diese Vorlesung ist Dennis M. Ritchie gewidmet, der am 12. Oktober 2011 verstorben ist und ohne dessen Beiträge diese Vorlesung in dieser Form nicht denkbar wäre. Dennis M. Ritchie hat nicht nur die Programmiersprache C entworfen und implementiert, sondern auch zusammen mit Ken Thompson Unix entwickelt. Mit C wurde erstmals eine höhere Programmiersprache statt Assembler für die Systemprogrammierung verwendet. Dennis Ritchie und seinen Kollegen gelang es bei Unix, die Betriebssystemsschnittstellen und die Systemwerkzeuge in einem bis dahin unbekanntem Maß zu vereinfachen. Auf sie geht die Umsetzung des wichtigen Grundsatzes zurück, dass jedes Werkzeug genau eine Aufgabe zu erfüllen habe und das möglichst gut.

Inhalte:

- Einführung in die Programmiersprache C
- Dynamische Speicherverwaltung
- Entwicklungswerkzeuge im Umfeld von C
- Dateisysteme
- Systemnahe Programmierung

Syllabus: Ziele

- Erwerb von Grundkenntnissen der Programmiersprache C, wobei ein besonderer Wert gelegt wird auf den Umgang mit der dynamischen Speicherverwaltung und mit den Zeigern in C. Ziel ist es auch, den versehentlichen Einbau von Sicherheitslücken zu vermeiden.
- Erlernen des Umgangs mit den klassischen Entwicklungswerkzeugen unter UNIX wie beispielsweise make.
- Verständnis der Abstraktion eines Dateisystems, einiger Implementierungen und praktische Erfahrungen mit der zugehörigen System-Schnittstelle.

- Schriftliche Prüfungen gibt es am 18. Februar und 2. April 2018. Die genaue Uhrzeit und die Räume stehen noch nicht fest. Die Prüfungen sind offen.
- Zur Teilnahme an der Prüfung ist keine Vorleistung mehr erforderlich.
- Ohne eine intensive Teilnahme an den Übungen ist jedoch eine erfolgreiche schriftliche Prüfung eher unrealistisch.
- Der Umfang beträgt 6 Leistungspunkte.
- Studiengänge:
  - **Bachelor:** CSE, Mathematik, Wirtschaftsmathematik, Informatik, Medieninformatik, Software Engineering, Physik, Wirtschaftsphysik und Elektrotechnik.
  - **Master:** Informatik und Medieninformatik (Technische und Systemnahe Informatik).

Syllabus: Voraussetzungen

- Grundkenntnisse in Informatik. Insbesondere sollte keine Scheu davor bestehen, etwas zu programmieren.
- Freude daran, etwas auch an einem Rechner auszuprobieren und genügend Ausdauer, dass nicht beim ersten Fehlversuch aufgegeben wird.

- Jede Woche gibt es zwei Vorlesungsstunden an jedem Donnerstag von 16-18 Uhr im H12.
- Die Übungen unter Leitung von Moritz Carmesin finden am Freitag von 14-16 Uhr im H14 statt.
- Organisatorische Feinheiten werden in der ersten Übungsstunde erläutert.
- Webseite: `https://www.uni-ulm.de/mawi/mawi-numerik/lehre/wintersemester-20182019/vorlesung-systemnahe-software-i/`
  Oder kürzer: `https://tinyurl.com/y8slz4ko`

Syllabus: Übungen

- Wer an den Übungen teilnehmen möchte, der sollte sich über SLC für die Vorlesung registrieren.
- Eine Registrierung bei SLC ist völlig unverbindlich.
- Die Lösungen zu den Übungen werden elektronisch eingereicht. Das eröffnet uns die Möglichkeit, die Lösungen teilweise automatisiert auszuwerten und/oder als Grundlage für Feedback in den Übungen zu nehmen.
- Ebenso sollten alle über einen Shell-Zugang zu unseren Servern (wie z.B. die Theon) verfügen.
- Einzelheiten werden in der ersten Übungsstunde am Freitag, den 19. Oktober, um 14 Uhr im H14 vorgestellt.

- Es gibt ein Skript (entwickelt von mehreren Mitgliedern meines ehemaligen Instituts), das auf der Vorlesungswebseite zur Verfügung steht.
- Das Skript wird schrittweise entsprechend dem Vorlesungsverlauf zur Verfügung gestellt werden.
- Parallel gibt es Präsentationen (wie diese), die ebenfalls als PDF zur Verfügung gestellt werden.

Syllabus: Sprechstunden

- Sie sind eingeladen, mich jederzeit per E-Mail zu kontaktieren:
  E-Mail: andreas.borchert@uni-ulm.de
- Meine reguläre Sprechzeit ist am Mittwoch 10:00-11:30 Uhr. Zu finden bin ich in der Helmholtzstraße 20, Zimmer 1.23.
- Zu anderen Zeiten können Sie auch gerne vorbeischauen, aber es ist dann nicht immer garantiert, daß ich Zeit habe. Gegebenenfalls lohnt sich vorher ein Telefonanruf: 23572.

Syllabus: Nicht verzweifeln!

- Immer wieder kann es mal vorkommen, dass es zu scheinbar unlösbaren Problemen bei einer Übungsaufgabe kommt.
- Geben Sie dann bitte nicht auf. Nutzen Sie unsere Hilfsangebote.
- Sie können dazu gerne den Übungsleiter Moritz Carmesin oder auch mich kontaktieren.
- Schicken Sie bitte in so einem Fall alle Quellen zu und vergessen Sie nicht, eine präzise Beschreibung des Problems mitzuliefern.
- Das kann auch am Wochenende funktionieren.

Syllabus: Feedback

- Feedback ist ausdrücklich erwünscht.
- Es besteht insbesondere auch immer die Möglichkeit, auf Punkte noch einmal einzugehen, die zunächst noch nicht klar geworden sind.
- Vertiefende Fragen und Anregungen sind auch willkommen.
- Wir spulen hier nicht immer das gleiche Programm ab. Jede Vorlesung und jedes Semester verläuft anders und das hängt auch von Ihnen ab!

Syllabus: Wie geht es weiter?

- SS 2019 folgte der zweite Teil der Vorlesung (Systemnahe Software II).
- Ebenfalls im SS 2019 werde ich auch die Vorlesung Parallele Programmierung mit C++ anbieten.
- Im SS 2020 wird Objektorientierte Programmierung mit C++ angeboten werden.

- Der Begriff „System“ bezieht sich hier auf den Kern eines Betriebssystems.
- Betriebssysteme (bzw. deren Kerne) erfüllen drei Funktionen:
  - Sie greifen direkt auf die Hardware zu,
  - sie verwalten all die Hardware-Ressourcen wie beispielsweise Speicherplatz, Plattenplatz und CPU-Zeit und
  - sie bieten eine Schnittstelle für Anwendungsprogramme.
- Systemnahe Software ist Software, die direkt mit der Betriebssystems-Schnittstelle zu tun hat.

Wie sehen die Schnittstellen aus?

- Teilweise bieten die Betriebssystems-Schnittstellen (auch Systemaufrufe genannt) ein sehr hohes Abstraktions-Niveau.
- So kann beispielsweise aus der Sicht einer Anwendung eine Netzwerk-Kommunikation abgewickelt werden, ohne darüber nachzudenken, was für Netzwerk-Hardware konkret genutzt wird, wie die Pakete geroutet werden oder wann Pakete erneut zu senden sind, wenn der erste Versuch nicht geklappt hat.
- Zwar gibt es teilweise große Unterschiede bei den Schnittstellen, jedoch steht erfreulicherweise ein Standard zur Verfügung, kurz POSIX genannt oder ausführlicher IEEE Standard 1003.1-2017.
- Dieser Standard entspricht weitgehend einer gemeinsamen Schnittmenge von Unix, Linux und den BSD-Varianten. Dank Cygwin gibt es auch weitgehend eine POSIX-Schnittstelle unter Windows. Seit Windows 10 gibt es auch ein *Windows Subsystem for Linux*, das über POSIX hinaus auch die gesamte Schnittstelle des Linux-Kernels anbietet.

Warum ist C relevant?

- Durch den Erfolg von Unix erreichte C eine gewisse Monopolstellung für die systemnahe Software.
- Da POSIX (und auch die einzelnen Betriebssysteme) die Schnittstelle nur auf der Ebene von C definieren, führt an C kaum ein Weg vorbei.
- Praktisch alle anderen Sprach-Implementierungen (wie beispielsweise C++, Java, Fortran oder Ada) basieren letztenendes auf C bzw. benötigen die C-Bibliothek, die die Schnittstelle zum Betriebssystem liefert.

Warum überhaupt etwas anderes als C? 17

- Ursprünglich wurde systemnahe Software in Assembler geschrieben. Das ist sehr umständlich, fehleranfällig und überhaupt nicht portabel, da jede Prozessor-Architektur vollkommen anders zu programmieren ist.
- C entstand Mitte der 70er Jahre als Alternative zu Assembler. Manche bezeichnen C deswegen bis heute als „portablen Assembler“.
- C liefert Portabilität, ist aber immer noch sehr maschinennah.
- Wir verlassen mit C die gewohnte „heile Welt“ von Java (oder anderer ähnlicher moderner Programmiersprachen).
- C setzt maschinennahes Denken voraus und bietet viele Fallstricke, die wir in der „heilen Welt“ nicht kennen.
- Insofern wird heute C bevorzugt nur im systemnahen Bereich eingesetzt, wo diese Freiheiten benötigt werden.

Wie funktionieren Systemaufrufe? 18

- Programme laufen auf modernen Betriebssystemen in ihrer eigenen virtuellen Welt ab, d.h. sie sehen in ihrem Adressraum weder das Betriebssystem noch die anderen parallel laufenden Programme.
- Die virtuelle Welt wird nur durch besondere Ereignisse verlassen, wenn z.B. durch 0 geteilt wird, ein Nullzeiger dereferenziert wird, die Uhr sagt, dass ein anderer Prozess mal an der Reihe ist, sich die Platte meldet, weil ein gewünschter Datenblock endlich da ist, irgendeine Taste auf der Tastatur gedrückt wurde oder ...
- ... ein Programm mit dem Betriebssytem kommunizieren möchte.
- All diese Ereignisse unterbrechen die Ausführung eines Benutzerprogramms und übergeben die Kontrolle dem Betriebssystem-Kern, der dann zu untersuchen hat, was die Unterbrechung auslöste.
- Im Falle eines Systemaufrufs werden die Parameter aus der Welt des Benutzerprogramms mühsam herausgeholt, der Aufruf bearbeitet und die Resultate in die Benutzer-Welt überführt.
- In Wirklichkeit ist das noch viel komplizierter ...

Wie wird eine Unterbrechung initiiert? 19

- Für absichtliche Unterbrechungen gibt es spezielle Maschinen-Instruktionen.
- Diese gehören nicht zum Vokabular eines C-Compilers, so dass in jeder C-Bibliothek die Systemaufrufe in Assembler (bzw. mit in C eingebetten Assembler-Instruktionen) geschrieben sind.
- Die Aufrufsyntax in C ist portabel, die jeweilige Implementierung ist es nicht, da Assembler-Programme von der jeweiligen Plattform (Maschinenarchitektur und Betriebssystem) abhängig sind.

Wie sehen Systemaufrufe konkret aus?

hello.s

```
/*
   Hello world demo in Assembler
   for the SPARCv8/Solaris platform
*/
        .section ".text"
        .globl  _start
_start:
/* write(1, msg, 13); */
        or      %g0,4,%g1
        or      %g0,1,%o0
        sethi   %hi(msg),%o1
        add     %o1,%lo(msg),%o1
        or      %g0,13,%o2
        ta      8
/* exit(0) */
        or      %g0,1,%g1
        or      %g0,0,%o0
        ta      8
msg:    .ascii  "Hello world!\012"
```

- Das Beispiel wurde für die SPARC-Architektur geschrieben.
- Das Assembler-Programm besteht aus 9 Instruktionen, die jeweils 4 Bytes benötigen und 13 Bytes Text.
- `%g1`, `%o0`, `%o1` und `%o2` sind alles sogenannte Register, die im 32-Bit-Modus jeweils 32 Bit aufnehmen können.
- *%g0* ist ein spezielles Register, das immer den Wert 0 hat.
- Instruktionen haben bei der SPARC-Architektur normalerweise drei Operanden, wobei der dritte Operand das Ziel ist. Beispiel: `or %g0,4,%g1`
  Das ist eine binäre Oder-Operation mit *%g0* (also dem Wert 0) und der Zahl 4, dessen Resultat in *%g1* abgelegt wird. Kurz gefasst wird damit dem Register *%g1* der Wert 4 zugewiesen.

Wie sehen Systemaufrufe konkret aus? 22

- Die spezielle Instruktion *ta* (*trap always*) unterbricht die Programmausführung, bis der Prozess vom Betriebssystem wieder zum Leben erweckt wird.
- Die Parameter des Systemaufrufs werden bei der SPARC/Solaris-Plattform in den Registern *%o0* bis *%o5* abgelegt (bis zu 6 Parameter, die allerdings auf irgendwelche Speicherflächen mit mehr Parametern verweisen können).
- Im Register *%g1* wird eine Nummer abgelegt, die den Systemaufruf selektiert. So steht beispielsweise die 1 für *exit()* und 4 für *write()*. (Die Nummern finden sich unter Solaris in */usr/include/sys/syscall.h*.)
- Die Nummer 8, die bei *ta* angegeben wird, dient als Index in die Trap-Tabelle ...

Was passiert nach einem Systemaufruf? 23

usr/src/uts/sun4u/ml/trap_table.s

```
trap_table:
    /* hardware traps */
    NOT;                        /* 000  reserved */
    RED;                        /* 001  power on reset */
    RED;                        /* 002  watchdog reset */
    RED;                        /* 003  externally initiated reset */
/* ... */
    /* user traps */
    GOTO(syscall_trap_4x);      /* 100  old system call */
    TRAP(T_BREAKPOINT);         /* 101  user breakpoint */
    TRAP(T_DIV0);               /* 102  user divide by zero */
    FLUSHW();                   /* 103  flush windows */
    GOTO(.clean_windows);       /* 104  clean windows */
    BAD;                        /* 105  range check ?? */
    GOTO(.fix_alignment);       /* 106  do unaligned references */
    BAD;                        /* 107  unused */
    SYSCALL(syscall_trap32)     /* 108  ILP32 system call on LP64 */
/* ... */
```

- (Der Programmtext stammt aus den OpenSolaris-Quellen, ursprünglich von `http://www.opensolaris.org/`, jetzt verfügbar in revidierter Form über `http://src.illumos.org`).

Was passiert nach einem Systemaufruf? 24

usr/src/uts/sun4u/ml/trap_table.s

```
#define SYSCALL(which)                  \
        TT_TRACE(trace_gen)             ;\
        set     (which), %g1            ;\
        ba,pt   %xcc, sys_trap          ;\
        sub     %g0, 1, %g4             ;\
        .align  32
```

- Zunächst werden alle Register gesichert und es findet ein Wechsel in den privilegierten Prozessor-Modus statt.
- Zu jeder Unterbrechungsart gibt es eine Nummer, wobei Unterbrechungen durch Benutzerprogramme von Hardware-Unterbrechungen unterschieden werden. Aus *ta 8* wird die Nummer $256 + 8 = 0\text{x}108$.
- Zu jedem der 512 verschiedenen Unterbrechungsmöglichkeiten sind in der Trap-Tabelle 32 Bytes Code vorgesehen, die den Trap behandeln, indem sie typischerweise eine entsprechende Routine aufrufen.

Was passiert nach einem Systemaufruf? 25

usr/src/uts/sparc/v9/ml/syscall_trap.s

```
    ENTRY_NP(syscall_trap32)
    ldx     [THREAD_REG + T_CPU], %g1     ! get cpu pointer
    mov     %o7, %l0                      ! save return addr
/* ... */
    lduw    [%l1 + G1_OFF + 4], %g1       ! get 32-bit code
    set     sysent32, %g3                 ! load address of vector table
    cmp     %g1, NSYSCALL                 ! check range
    sth     %g1, [THREAD_REG + T_SYSNUM] ! save syscall code
    bgeu,pn %ncc, _syscall_ill32
      sll   %g1, SYSENT_SHIFT, %g4        ! delay - get index
    add     %g3, %g4, %g5                 ! g5 = addr of sysentry
    ldx     [%g5 + SY_CALLC], %g3         ! load system call handler
/* ... */
    call    %g3                           ! call system call handler
    nop
/* ... */
    jmp     %l0 + 8
     nop
```

- Danach werden die Parameter so kopiert, dass sie als Parameter einer C-Funktion übergeben werden können (nicht dargestellt) und dann wird passend zur Systemaufrufsnummer die entsprechende Funktion aus einer Tabelle ausgewählt.

Was passiert nach einem Systemaufruf? 26

usr/src/uts/common/os/sysent.c

```
struct sysent sysent[NSYSCALL] =
{
/* ONC_PLUS EXTRACT END */
    /*  0 */ IF_LP64(
                    SYSENT_NOSYS(),
                    SYSENT_C("indir",       indir,          1)),
    /*  1 */ SYSENT_CI("exit",              rexit,          1),
    /*  2 */ SYSENT_2CI("forkall",          forkall,        0),
    /*  3 */ SYSENT_CL("read",              read,           3),
    /*  4 */ SYSENT_CL("write",             write,          3),
    /*  5 */ SYSENT_CI("open",              open,           3),
```

- In der *sysent*-Tabelle finden sich alle Systemaufrufe zusammen mit einigen Infos zur Anzahl der Parameter und die Art der Rückgabewerte. Unter Solaris hat diese Tabelle inzwischen 256 Einträge.

Was passiert nach einem Systemaufruf? 27

usr/src/uts/common/syscall/rw.c

```
ssize_t write(int fdes, void *cbuf, size_t count) {
        struct uio auio;
        struct iovec aiov;
/* ... */
        if ((cnt = (ssize_t)count) < 0)
                return (set_errno(EINVAL));
        if ((fp = getf(fdes)) == NULL)
                return (set_errno(EBADF));
        if (((fflag = fp->f_flag) & FWRITE) == 0) {
                error = EBADF;
                goto out;
        }
/* ... */
        aiov.iov_base = cbuf;
        aiov.iov_len = cnt;
/* ... */
        auio.uio_loffset = fileoff;
        auio.uio_iov = &aiov;
/* ... */
        error = VOP_WRITE(vp, &auio, ioflag, fp->f_cred, NULL);
        cnt -= auio.uio_resid;
/* ... */
out:
/* ... */
        if (error)
                return (set_errno(error));
        return (cnt);
}
```

Weiterleitung der Schreib-Operation

usr/src/uts/common/sys/vnode.h

```
#define VOP_WRITE(vp, uiop, iof, cr, ct) \
        fop_write(vp, uiop, iof, cr, ct)
```

usr/src/uts/common/fs/vnode.c

```
int
fop_write(vnode_t *vp, uio_t *uiop, int ioflag, cred_t *cr,
        caller_context_t *ct)
{
        int     err;
        ssize_t resid_start = uiop->uio_resid;

        VOPXID_MAP_CR(vp, cr);

        err = (*(vp)->v_op->vop_write)(vp, uiop, ioflag, cr, ct);
        VOPSTATS_UPDATE_IO(vp, write,
            write_bytes, (resid_start - uiop->uio_resid));
        return (err);
}
```

- Hier wird ein Funktionszeiger aufgerufen – das entspricht einem dynamischen Methodenaufruf.

Wie sehen Systemaufrufe unter Linux aus? 29

hello-x86.s

```
/*
 * Hello world demo in Assembler
 * for the IA-64/Linux platform
 */
        .text
        .globl  _start
_start:
/* write(1, msg, 13) */
        movl    $13, %edx
        movl    $msg, %esi
        movl    $1, %edi
        movl    $1, %eax
        syscall

/* exit(0) */
        movl    $0, %edi
        movl    $60, %eax
        syscall

        .data
msg:
        .string "Hello world!\n"
```

Wie sehen Systemaufrufe unter Linux aus? 30

- Auf der IA-64/Linux-Plattform werden die Parameter ebenfalls über Register übermittelt. Hier die ersten drei Parameter über *%edi*, *%esi* und *%edx*.
- Die Systemaufrufnummer wird in *%eax* abgelegt. Linux verwendet jedoch seine eigene Numerierung mit 1 für *write()* und 60 für *edit()*.
- Der Trap für einen Systemaufruf wird auf der IA-64-Plattform mit der Instruktion *syscall* initiiert.

```
heim$ as -o hello-x86.o hello-x86.s
heim$ ld -o hello-x86 hello-x86.o
heim$ ./hello-x86
Hello world!
heim$ strace ./hello-x86 >/dev/null
execve("./hello-x86", ["./hello-x86"], [/* 46 vars */]) = 0
write(1, "Hello world!\n", 13)           = 13
_exit(0)                                 = ?
+++ exited with 0 +++
heim$
```

- Mit *as* wird der Assemblertext in Maschinencode überführt. Das Resultat ist eine sogenannte Objektdatei mit der Endung „.o“.
- Objekt-Dateien können mit dem Werkzeug *ld* (*loader* oder *linkage editor*) zu einem ausführbaren Programm zusammengefügt werden, ggf. unter Verwendung von Bibliotheken.
- Mit dem Werkzeug *strace* können die Systemaufrufe eines Prozesses verfolgt werden (unter Solaris ist hier *truss* zu verwenden).

Zusammenfassung

- Die Schnittstelle zwischen Anwendungen und dem Betriebssystems-Kern besteht aus über 200 einzelnen Funktionen, die zu einem großen Teil standardisiert sind.
- Systemaufrufe sind sehr viel teurer als reguläre Funktionsaufrufe. Das liegt an dem Mechanismus der Unterbrechungsbehandlung, dem notwendigen Kontextwechsel, der Parameterschaufelei und auch der asynchronen Natur vieler Funktionen (wie beispielsweise beim I/O).
- Deswegen ist es wichtig, auf der Anwendungsseite Bibliotheken zu entwickeln, die die Zahl der Systemaufrufe bzw. deren Aufwand minimieren.
- Verbunden mit den einzelnen Systemaufrufen sind viele Abstraktionen und Objekte, die wir uns im Laufe der Vorlesung genauer ansehen werden wie etwa das Dateisystem, die Prozesse, die Signalbehandlung (Unterbrechungen auf der Benutzerseite) die Interprozess-Kommunikation und die allgemeine Netzwerk-Kommunikation.