4. Vom C-Programm zum laufenden Prozess

Michael Schöttner

Betriebssysteme und Systemprogrammierung

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

4.1 Vorschau

- Vom C-Programm zum ausführbaren Programm (engl. executable)
  - Präprozessor
  - Compilieren
  - (Assemblieren)
  - Binden (statisch / dynamisch)

- Programm und Prozess
  - Speicherorganisation eines Programms
  - Speicherorganisation eines Prozesses
  - Laden eines Programms (statisch gebunden / dynamisch gebunden

- Prozesszustände

4.2 Übersetzen

- 1. Schritt: Präprozessor
  - entfernt Kommentare, wertet Präprozessoranweisungen aus
    - fügt include-Dateien ein
    - expandiert Makros
    - entfernt ggf. Makro-abhängige Code-Abschnitte (conditional code)

Beispiel:

```
#define DEBUG 1
...
#ifdef DEBUG
   printf("Zwischenergebnis = %d\n", wert);
#endif DEBUG
```

  - Zwischenergebnis kann mit **`cc -E datei.c`** ausgegeben werden

Objektmodule

- 2. Schritt: Compilieren
  - übersetzt C-Code in Assembler
  - wenn Assemblercode nicht explizit angefordert wird, direkter Übergang zu Schritt 3.
  - Zwischenergebnis kann mit **`cc -save-temps datei.c`** als **`datei.s`** erzeugt werden (AT&T Assembly Syntax)

```
_main:                          ## @main
        .cfi_startproc
## BB#0:
        pushq   %rbp
Lcfi0:
        .cfi_def_cfa_offset 16
Lcfi1:
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
Lcfi2:
        .cfi_def_cfa_register %rbp
        subq    $16, %rsp
        leaq    L_.str(%rip), %rdi

        ...
```

Objektmodule

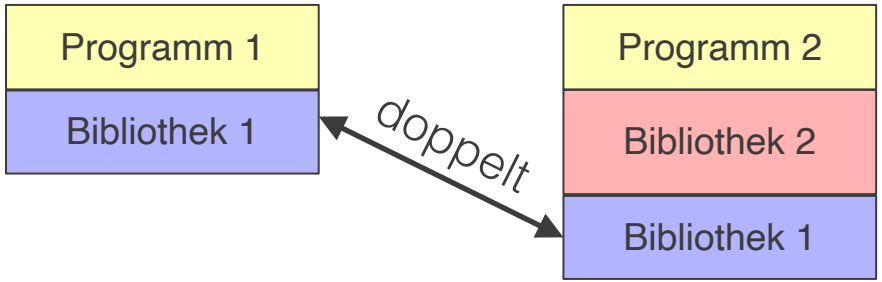
- 3. Schritt: Assemblieren
  - Assembler-Code nach Maschinencode (Objekt-Datei) assemblieren
  - Erzeugt in UNIX Objektdatei: Executable and Linking Format (ELF)
    in anderen Betriebssystemen andere Formate
    - Maschinencode
    - Informationen über Variablen mit Lebensdauer *static*
      (ggf. Initialisierungswerte)
    - Symboltabelle: wo stehen welche globale Variablen und Funktionen
    - Relocation-Information: Referenzen auf globale Variablen bzw. Funktionen
      in anderen Modulen
  - Zwischenergebnis kann mit `cc -c datei.c` als `datei.o` erzeugt werden
  - Weitere Infos: https://wiki.osdev.org/ELF

Binden

- 4. Schritt: Binden
  - Linker `ld` erzeugt ausführbare Datei (engl. executable file)
    - ebenfalls ELF-Format (früher a.out-Format oder COFF)
  - Objekt-Dateien (.o-Dateien) werden zusammengebunden
    - Modulübergreifende Referenzen auf globale Variablen und Funktionen werden gebunden (Relocation-Information)
  - fehlende Funktionen und globale Variablen werden in Bibliotheken gesucht

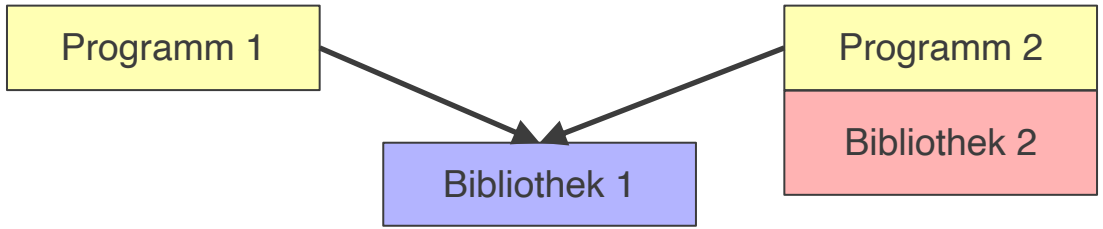
Bibliotheken

- statisch binden
  - alle fehlenden Funktionen werden aus Bibliotheken genommen und in die ausführbare Datei einkopiert
  - i.d.R. wird die gesamte statische Bibliothek (engl. static library) einkopiert
    - Ausführbare Datei wird ggf. groß
    - Bibliothek wird ggf. mehrfach geladen, wenn mehrere Programme gleichzeitig ausgeführt werden, die die gleiche statische Bibliothek verwenden
    - Vorteil: Der Fehler, dass eine Bibliothek fehlt wird vermieden



Bibliotheken

- dynamisch binden
  - Eine gemeinsam nutzbare Bibliothek (engl. shared library) wird nicht in die ausführbare Datei einkopiert, sondern eingeblendet
    - Benötigte Funktionen werden durch eine Indirektion referenziert
    - Dadurch ist sind ausführbare Dateien kleiner
    - Mehrfach genutzte Bibliotheken werden auch nur ein Mal geladen
  - Auflösung der Referenzen (Relokation) erfolgt beim Laden
  - Nachteil: Wird ein Bibliothek versehentlich gelöscht, so kann Programm nicht mehr ausgeführt werden



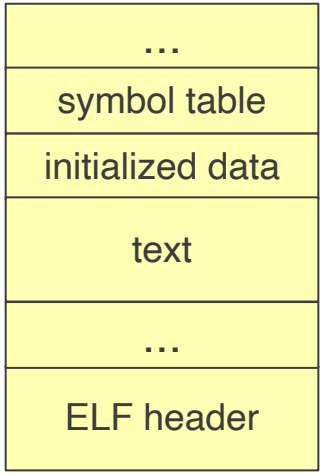
4.3 Begriffe: Programme und Prozesse

- **Programm**: Folge von Anweisungen
  (ausführbare Datei auf dem Hintergrundspeicher)

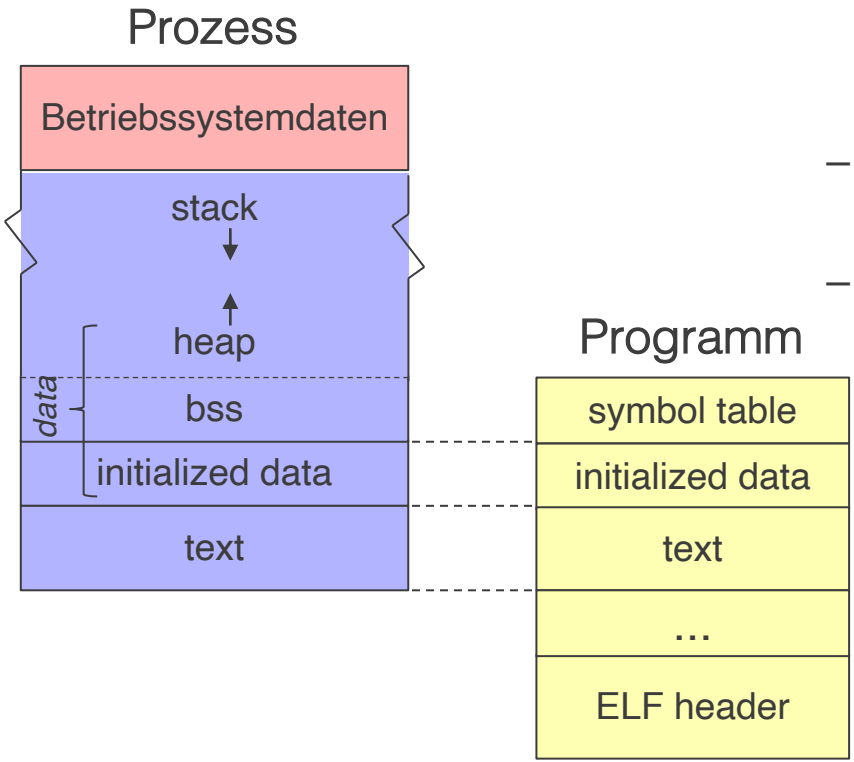
- **Prozess**: Programm, das sich in Ausführung befindet, und seine Daten
  - Zu einem Prozess gehört immer Speicher, Rechte und Verwaltungsinformationen
  - Evt. wird ein Programm gleichzeitig mehrfach ausgeführt

4.4 Speicherorganisation eines Programms

- Definiert durch das ELF-Format (= Executable and Linking Format)
- Wichtigste **Segmente** (vereinfach dargestellt)
  - **ELF header**: Identifikator und Verwaltungsinformationen, u.a. Dateityp:
    - Relocatable object file: `.o` Datei
    - Executable object file: `a.out` Datei
    - Shared object file: `.so` Datei
  - **text** Programmcode
  - **initialized data** initialisierte globale und static Variablen
  - **symbol table** Zuordnung der im Programm verwendeten symbolischen Namen von Funktionen und globalen Variablen zu Adressen (z. B. für Debugger)



4.5 Speicherorganisation eines Prozesses



- **bss** nicht initialisierte globale und `static` Variablen (wird vor der Vergabe an den Prozess mit 0 vorbelegt)
- **heap** dynamische Erweiterungen des `bss`-Segments
- **stack** lokale Variablen, Funktionsparameter, Speicherbereiche für Registerinhalte, (wird bei Bedarf dynamisch erweitert)

4.6 Laden eines Programms

- In eine Ausführungsumgebung (Prozess) kann ein Programm geladen werden.
  → Dies erledigt der Lader (engl. loader)

- Laden statisch gebundener Programme
  - Segmente der ausführbaren Datei werden in den Speicher geladen
    - abhängig von der jeweiligen Speicherorganisation des Betriebssystems
  - Speicher für nicht-initialisierte globale und `static` Variablen (bss) wird bereitgestellt
  - Speicher für lokale Variablen (stack) wird bereitgestellt
  - Aufrufparameter werden in Stack- oder Datensegment kopiert,
    `argc` und `argv`-Zeiger werden entsprechend initialisiert
  - Und dann wird main-Funktion wird angesprungen

4.6 Laden eines Programms

- Laden dynamisch gebundener Programme
  - Spezielles Lade-Programm wird gestartet: **`ld.so`** (dynamic linker/loader)
  - `ld.so` erledigt die weiteren Aufgaben
    - Segmente der ausführbaren Datei werden in den Speicher geladen
    - Shared libraries werden bei Bedarf geladen und fehlende Funktionen eingebunden (ggf. rekursiv)
    - Offene Referenzen werden initialisiert
    - Wenn notwendig werden Initialisierungsfunktionen der shared libraries aufgerufen (z. B. Klasseninitialisierungen bei C++)
    - Dann weiter wie bei statisch gebundenen Programmen ...

Adressbindung zur Übersetzungszeit

- Fest „verdrahtete“ Adressen
- Konsequenz: Programm muss an bestimmte Adresse geladen werden
  - Technisch ist dies kein Problem, da jedes Programm als eigener Prozess läuft und somit imaginär den Speicher für sich alleine hat.
  - Aus Sicherheitsgründen ist es besser ein Programm nicht immer an die gleiche feste Adresse zu laden
- Beispiel: Aufruf einer Funktion in einer statisch gebunden Bibliothek →
  - Statische Libraries werden aber i.d.R. auch mit positionsunabh. Code erzeugt

```
0000000000400526 <main>:
  400526: push   rbp
  400527: mov    rbp,rsp
  40052a: mov    edi,0x04005e4
  40052f: call   40053b <hello>
  400534: mov    eax,0x0
  400539: pop    rbp
  40053a: ret

000000000040053b <hello>:
  40053b: push   rbp
  40053c: mov    rbp,rsp
             ...
```

Beispiel: staticlib

- Siehe Quelltext

- Eine statische Bibliothek ist eine Menge von Objektdateien, welche in eine Datei mit dem Suffix `.a` kopiert werden.
- Die statische Bibliothek wird mit dem archiver (`ar`) erzeugt:

Adressbindung zur Ladezeit

- **Shared Libraries** werden durch den Lader geladen (falls noch nicht für ein anderes Programm schon geladen)
- Damit der Lader flexibel bleibt, darf die Lade-Adresse der Shared-Library nicht festgelegt werden und der generierte Code muss positionsunabhängig sein.
- Ansonsten könnte es zu Konflikten bezüglich der Lade-Adressen kommen, wenn eine Shared-Library von mehreren Programmen verwendet wird.
- Sobald die Lade-Adresse feststeht, trägt der Lader die Adressen der Funktionen in eine Tabelle des aufrufenden Programms ein (siehe Beispiel auf der nächsten Seite)

Adressbindung zur Ladezeit (2)

- Hier ist nun ein Funktionsaufruf einer Funktion in einer Shared-Library.
- Die Indirektion über eine Tabelle fällt sofort auf
  (im Gegensatz zur statisch gebundenen Library)

```
          ...

0000000000400570 <hello@plt>:
  400570: jmp    QWORD PTR [rip+0x200aaa]    # 601020 <_GLOBAL_OFFSET_TABLE_+0x20>
          ...
0000000000400686 <main>:
  400686: push   rbp
  400687: mov    rbp,rsp
  40068a: mov    edi,0x400728
  40068f: call   400570 <hello@plt>
  400694: mov    eax,0x0
  400699: pop    rbp
  40069a: ret
          ...
```

Indirektion über Tabelle
Lader trägt hier die Adressen ein.
Sprung erfolgt mithilfe relativer Adressierung zum Instruction-Pointer (=RIP)

4.7 Shared libraries in UNIX

- Namenskonvention: `libHello.so.1.2.3`
  - Präfix `lib` zeigt an, dass es sich um eine Bibliothek handelt
  - Suffix `.so` zeigt an, dass dies eine shared library ist
  - Versionsnummer: `Major.Minor.Release`
    - Major: wird erhöht, wenn sich API ändert
    - Minor: neue Funktionen oder Bugfix; kompatibel zur alten Version
    - Release: Bugfix; kompatibel zur alten Version (optionale Nummer)

- Versionierung vermeidet Probleme wie bei Microsoft Windows DLLs (Dynamic Link Libraries)
  - Wurde aber inzwischen durch Assemblies dort auch verbessert

Versionsmanagement

- Symbolische Verweise (Datei die einen Pfad auf eine andere Datei speichert) erlauben einfache Upgrades auf neuere Versionen
- Funktionen in shared libraries müssen nicht explizit exportiert werden

- **Linker** verwendet ohne weitere Angaben immer die neueste Version
- **Lader**:
  - Stützt sich auf Major-Nummer → neuere Releases werden so automatisch verwendet
  - Sucht in Pfaden, welche in der Umgebungsvariablen `LD_LIBRARY_PATH` stehen
  - Falls nicht gefunden, in Standard-Speicherorten suchen: `/lib`, `/usr/lib`, ...
  - Suche nach Bibliotheken ist langsam → Cache
    → in `/etc/ld.so.cache` (verwaltet mit `ldconfig`).

Beispiel: Shared Library

- Quelltext der Library:

```c
/* hello.c - demonstrate library use. */
#include <stdio.h>

void hello(char *msg) {
   printf("libHello: msg\n");
}
```

```c
/* hello.h */

void hello(char *msg);
```

- Quelltext des Testprogramms:

```c
/* testlib.c - demonstrate function call to shared library */
#include "hello.h"

int main() {
   hello("hello world\n"););
   return 0;
}
```

Beispiel: Shared Library

- **Compilieren**: Bibliothek muss mit Flag **`–fPIC`** übersetzt werden
  - PIC = Position Independent Code
  - Lader kann Adresse an die die Bibliothek geladen wird frei wählen
  - Wichtig, da bei fest vergebenen Ladeadressen schnell Konflikte entstehen könnten, wenn ein Programm mehrere Bibliotheken verwendet

```
gcc -fPIC -c hello.c                    # compile library
```

- **Binden**: Es wird der Lib-Name mit vollständiger Versionsnummer erwartet und auch der **`soname`** (nur mit der Major-Versionsnummer)
  - **`soname`** ist ein Feld im Header der Shared-Object-Datei
  - Zeit Kompatibilität der API an

```
gcc -shared -Wl,-soname,libhello.so.1      # create library with soname
    -o libhello.so.1.0.0 hello.o -lc       # -lc link library
```

Beispiel: Shared Library

- Setzen des **symbolischen Links für den Lader**:

```
ln -sf libhello.so.1.0.0 libhello.so.1      # set link for loader
```

  - Ergebnis: `libhello.so.1 -> libhello.so.1.0.0`
  - `soname` wird hier ignoriert. Aber der Lader verwendet das `needed` Feld im Header des Programms um die richtige Bibliothek zu laden

- Setzen des **symbolischen Links für den `soname` für den Linker**

```
ln -sf libhello.so.1 libhello.so            # set up link for soname
```

  - Ergebnis: `libhello.so -> libhello.so.1`
  - `soname` wird nur beim Linken berücksichtigt, wenn ein Programm gegen diese Bibliothek gebunden wird

Beispiel: Shared Library

- Compilieren des Testprogramms

```
gcc -c testlib.c -o testlib.o          # compile test program
```

- Linken des Testprogramms
  - **`-lhello`** linkt gegen **`libhello.so`**
  - Wichtig: Pfad für Library mit **`-L`** angeben, wenn diese nicht in Systempfad (z.B. **`/usr/lib`**) installiert

```
gcc -o demo testlib.o -L. -lhello      # bind demo prg
                                       # (-L lib search path for linker)
```

- Ausführen des Testprogramms
  - Auch hier Pfad für Library berücksichtigen, damit Lader diese findet

```
LD_LIBRARY_PATH="." ./demo             # test program
```

Nützliche Befehle

- `ldd` print shared object dependencies
- `readelf` display information about ELF files
- `objdump` display information from object files
  - `soname` Anzeigen: `objdump -p libhello.so | grep SONAME`

```
SONAME          libhello.so.1
```

  - Abhängigkeiten: `objdump -p demo | grep NEEDED`

```
NEEDED          libhello.so.1
NEEDED          libc.so.6
```

Beispiel: Shared Library – Laden zur Laufzeit

```
#include <stdlib.h>
#include <stdio.h>
#include <dlfcn.h>  /* dynamic loading shared libs */

void (*hello)();    /* function pointer to library function */

int main() {
   void *handle;
   char *error;

   handle = dlopen ("./libhello.so.1", RTLD_LAZY); /* resolve symbols lazily */
   if (handle==0) {
      printf("%s\n", dlerror());
      exit(1);
   }

   /* continued on next slide */
```

Beispiel: Shared Library – Laden zur Laufzeit (2)

```
   hello = dlsym(handle, "hello"); /* resolve function */
   error = dlerror();
   if (error!=NULL) {
      printf("%s\n", dlerror());
      exit(1);
   }

   (*hello)(); /* call lib function */

   dlclose(handle); /* release shared lib */
}
```

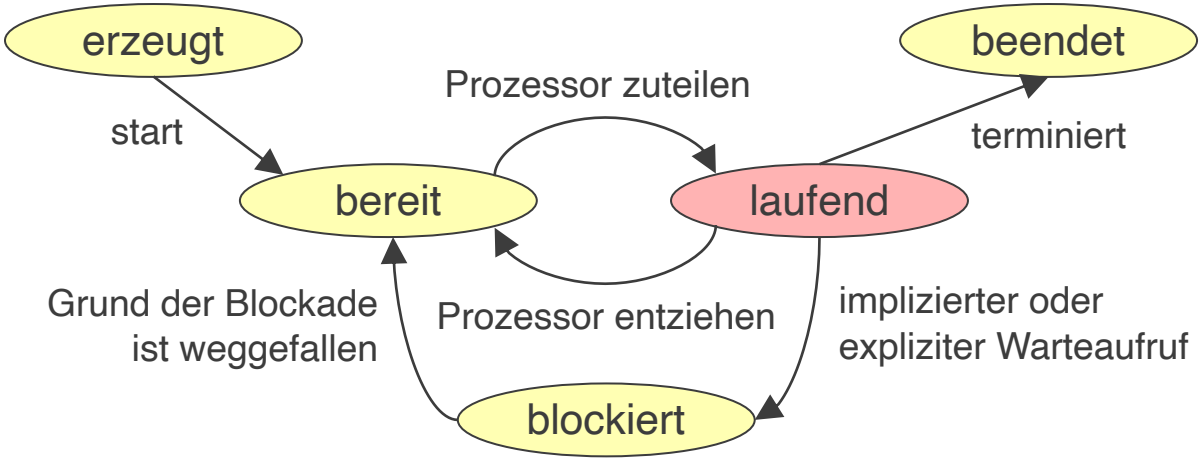
- Linken mit: **`cc –o demo dlib.c -ldl`**
- Laden zur Laufzeit wird heute selten verwendet.
- Ursprüngliche Motivation war Speicherersparnis

4.8 Prozesszustände

- Ein Prozess befindet sich in einem der folgenden Zustände:
  - **Erzeugt** (engl. created)
    Prozess wurde erzeugt, besitzt aber noch nicht alle nötigen Betriebsmittel
  - **Bereit** (engl. ready)
    Prozess besitzt alle nötigen Betriebsmittel und ist bereit zum Laufen
  - **Laufend** (engl. running)
    Prozess wird vom realen Prozessor ausgeführt
  - **Blockiert** (engl. blocked)
    Prozess wartet auf ein Ereignis (z.B. Abschluss einer Ein- oder Ausgabeoperation, Zuteilung eines Betriebsmittels, etc.); zum Warten wird er blockiert
  - **Beendet** (engl. terminated)
    Prozess ist beendet; einige Betriebsmittel sind aber noch nicht freigegeben oder Prozess muss aus anderen Gründen im System verbleiben

4.8 Prozesszustände

- Zustandsdiagramm



- **Scheduler** ist der Teil des Betriebssystems, der die Zuteilung des Prozessors regelt (siehe später)