Compiling and Linking

Praktikum „C-Programmierung“

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Eugen Betke, Nathanael Hübbe, Michael Kuhn, Jakob Lüttgau, Jannek Squar

2019-12-16

Wissenschaftliches Rechnen
Fachbereich Informatik
Universität Hamburg

GCC Compiler Toolchain: Eine Übersicht
- `binutils`
- GNU Compiler Collection
- Linux Kernel Headers
- Application Binary Interface (ABI)
- C Library

Compilation Process mit GCC

Compiler Flags
- Empfohlene Flags
- Beispiel: -fstack-protector
- Beispiel: -D_FORTIFY_SOURCE

Compiler Toolchain: Eine Übersicht

- `binutils`
- GCC: GNU Compiler Collection
  - C Library
  - Runtime
- Linux Kernel Headers

Alternative Toolchains:

- LLVM/Clang [a]
- Intel (C/C++, Fortran)
- Cray (C/C++, Fortran)
- IBM (C/C++)
- PGI
- NVIDEA (CUDA LLVM)
- AMD (AOCC: LLVM based)
- ARM (GCC or LLVM based)
- ...

[a] `https://clang.llvm.org/comparison.html`

binutils

Sammlung von *binary tools* [1]

as Assembler Create object files

ld Linker Combine object files/libs

addr2line Convert addresses into filenames/line numbers.

ar Create, modify and extract from archives.

c++filt Filter to demangle encoded C++ symbols.

dlltool Creates files for building and using DLLs.

gold A new, faster, ELF only linker, still in beta test.

gprof Displays profiling information.

nlmconv Converts object code into an NLM.

nm Lists symbols from object files.

objcopy Copies and translates object files.

objdump Displays information from object files.

ranlib Generates index to contents of an archive.

readelf Show information for ELF format object file.

size Lists section sizes of an object/archive file.

strings Lists printable strings from files.

strip Discards symbols.

[1] https://www.gnu.org/software/binutils/

GNU Compiler Collection

Frontends:

- C, C++, Objective-C, Fortran, Ada, Go, and D, (sogar Java bis GCC7/2016)

Backends:

- (70+) Architekturen/Plattformen

Bestandteile von GCC [2]:

cc1,cc1plus Compiler, erzeugt Assembly Code

gcc,g++ Compiler-Interface, Integration von `binutils`

Header-Files Deklarationen der Standard C-Library

libgcc, listdc++, libfortran Runtime-Libs

[2] `https://www.gnu.org/software/gcc/`

- Abstrahiert OS-Funktionalität über sog. System Calls
- Definition der Userspace-API in etwa 700 Header-Dateien (Linux 4.8)
- Nathlose Integration in GCC Projekt
- Die Header für die Runtime (aber auch Dritt-Bibliotheken) befinden sich bei den meisten Distributionen unter `/usr/include/`

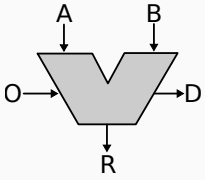
Application Binary Interface (ABI)

Das Application Binary Interface (ABI) definiert wie auf Datenstrukturen und Routinen in der Maschinenrepresentationen zugegriffen werden kann:

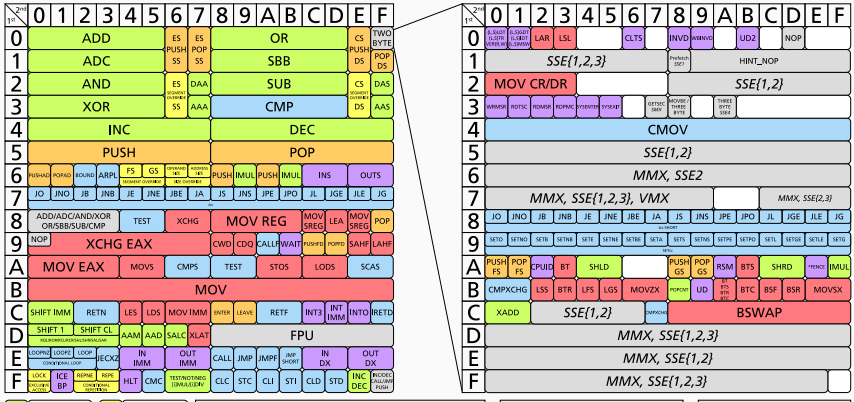
- Register Dateistruktur, Stack Organisation, Speicherlayout
- Größen, Aufbau und Alignments von Basistypen
- Aufruf-Konventionen: wie Argumente und Rückgabewerte übergeben werden
- Wie Systemaufrufe auszuführen sind
- Die Struktur der Object-Dateien

Die Linux ABI is weitesgehend rückwärtskompatibel:

- Daher, ältere Linux-Header i.d.R. weiterhin benutzbar:
  z.B. ein 3.4 Header funktioniert mit einem 4.5 Kernel
- Neue Header (insbesondere mit neuen Features) mit einem alten Kernel zu benutzen führt meistens zu Problemen



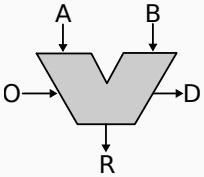
x86 Opcode Structure and Instruction Overview

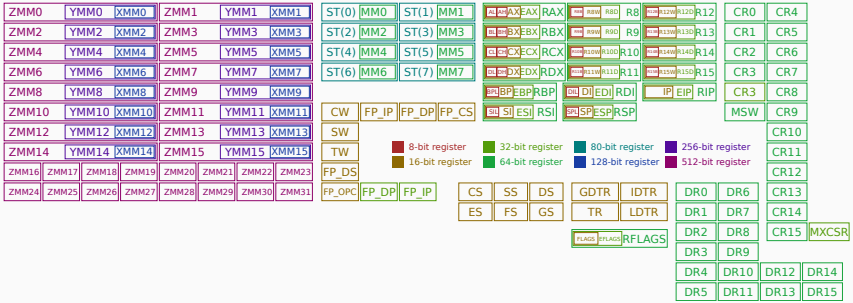


v1.0 – 30.08.2011
Contact: Daniel Plohmann – +49 228 73 54 228 – daniel.plohmann@fkie.fraunhofer.de

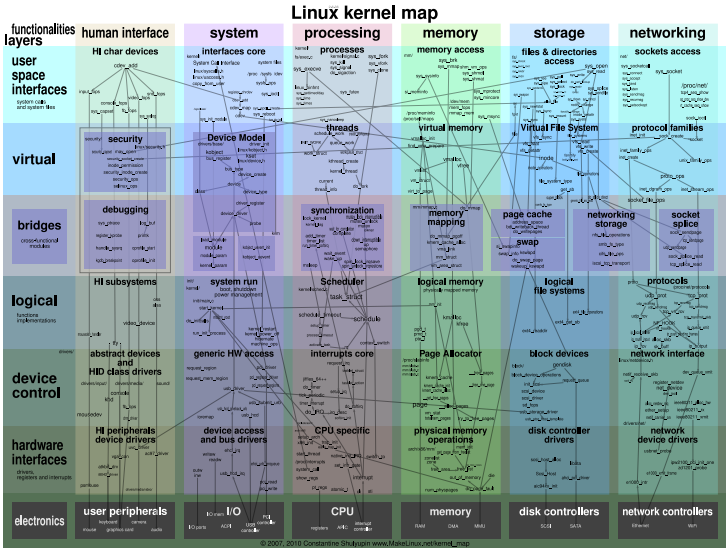
Source: Intel x86 Instruction Set Reference
Opcode table presentation inspired by work of Ange Albertini

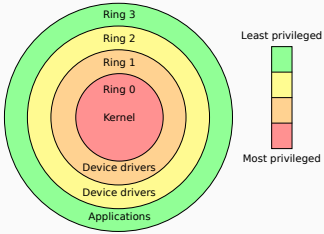
Hardware Perspektive (Registers)





Details: `https://en.wikipedia.org/wiki/X86#32-bit`





Details/Grafik:

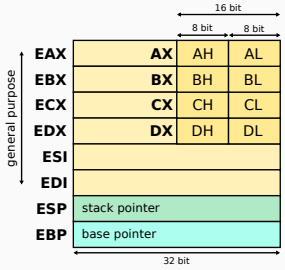
`https://en.wikipedia.org/wiki/Protected_mode`

Linux Kernel Map:

`https://makelinux.github.io/kernel/map/`

x86 Calling Conventions: cdecl

```c
int callee(int, int);
int caller(int a, int b, ..)
{
  return callee(1, 2) + 5;
}
```

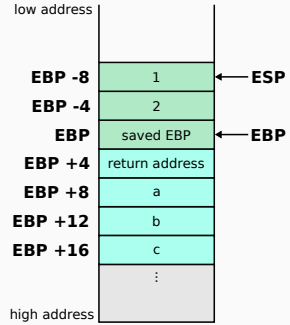



```asm
caller:
  ; make new call frame   (some compilers may produce an 'enter' instruction)
  push    ebp         ; save old call frame
  mov     ebp, esp    ; initialize new call frame
  ; push call arguments in reverse, or alternatively:
  ; sub esp, 8       : 'enter' instruction might do this for us
  ; mov [ebp-4], 2   : or mov [esp+8], 2
  ; mov [ebp-8], 1   : or mov [esp+4], 1
  push    2
  push    1
  call    callee      ; call subroutine 'callee'
  add     eax, 5      ; modify subroutine result
                      ; (eax is the return value of our callee, thus no extra mov)
  ; restore old call frame   (some compilers may produce a 'leave' instruction)
  ; add   esp, 8      ; remove arguments from frame, ebp - esp = 8.
                      ; compilers will usually produce the following instead,
                      ; which unlike the add instruction, also works for variable
                          ↪ length arguments/arrays
  mov     esp, ebp    ; most calling conventions dictate ebp be callee-saved
                      ; -> make sure the callee doesn't modify (or restores) ebp:
  pop     ebp         ; restore old call frame
  ret                 ; return
```

vgl. Grafik: http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html Details: https://en.wikipedia.org/wiki/X86_calling_conventions

x86 Calling Conventions: cdecl

```c
int callee(int, int);
int caller(int a, int b, ..)
{
  return callee(1, 2) + 5;
}
```




```asm
caller:
  ; make new call frame   (some compilers may produce an 'enter' instruction)
  push    ebp         ; save old call frame
  mov     ebp, esp    ; initialize new call frame
  ; push call arguments in reverse, or alternatively:
  ; sub esp, 8       : 'enter' instruction might do this for us
  ; mov [ebp-4], 2   : or mov [esp+8], 2
  ; mov [ebp-8], 1   : or mov [esp+4], 1
  push    2
  push    1
  call    callee      ; call subroutine 'callee'
  add     eax, 5      ; modify subroutine result
                      ; (eax is the return value of our callee, thus no extra mov)
  ; restore old call frame   (some compilers may produce a 'leave' instruction)
  ; add   esp, 8      ; remove arguments from frame, ebp - esp = 8.
                      ; compilers will usually produce the following instead,
                      ; which unlike the add instruction, also works for variable
                          ↪ length arguments/arrays
  mov     esp, ebp    ; most calling conventions dictate ebp be callee-saved
                      ; -> make sure the callee doesn't modify (or restores) ebp:
  pop     ebp         ; restore old call frame
  ret                 ; return
```

vgl. Grafik: https://eli.thegreenplace.net/2011/02/04/where-the-top-of-the-stack-is-on-x86/

Stellt POSIX Standard Funktionen bereit, und versteckt an vielen Stellen z.B. die Interaktion mit low-level Linux Systemcalls.

Es git diverse Implementationen der C Library jeweils mit unterschiedlichen Schwerpunkten:

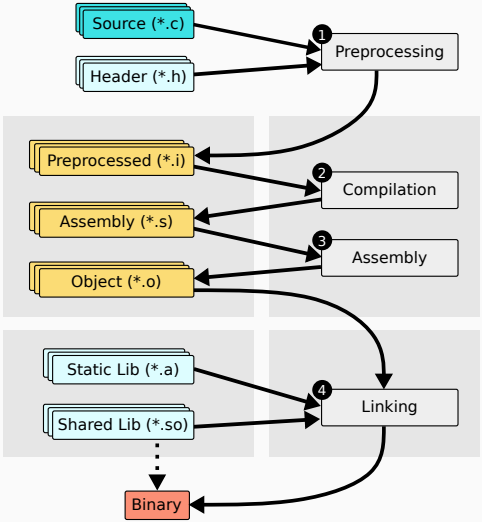
- glibc
- uClibc-ng
- musl
- bionic (Android)
- newlib, dietlib, klibc (for very minimal systems)

Beispiel: glibc Source Code: `https://sourceware.org/git/gitweb.cgi?p=glibc.git;a=tree;f=sysdeps/x86_64/nptl;hb=HEAD`

GCC Compiler Toolchain: Eine Übersicht
- binutils
- GNU Compiler Collection
- Linux Kernel Headers
- Application Binary Interface (ABI)
- C Library

Compilation Process mit GCC

Compiler Flags
- Empfohlene Flags
- Beispiel: -fstack-protector
- Beispiel: -D_FORTIFY_SOURCE



Ablauf [a]

1. `cpp hello.c > hello.i`
2. `gcc -S hello.i`
3. `as -o hello.o hello.s`
4. `ld -o hello hello.o -lc ...`

[a] Mehr Details: `gcc -v -o hello hello.c`

GCC Compiler Toolchain: Eine Übersicht
- `binutils`
- GNU Compiler Collection
- Linux Kernel Headers
- Application Binary Interface (ABI)
- C Library

Compilation Process mit GCC

Compiler Flags
- Empfohlene Flags
- Beispiel: -fstack-protector
- Beispiel: -D_FORTIFY_SOURCE

Compiler Flags

GCC kommt mit vielen Optionen und Einstellungen die i.d.R. über sog. Compiler-Flags gesteuert werden.

```
1 -Wl  werden an den linker (ld) weitergereicht (siehe "man ld")
```

```
# Sicherheit
-D_FORTIFY_SOURCE=2            Laufzeit Overflow Erkennung
-fpie -Wl,-pie                 Address Space Layout Randomization (ASLR)
-fstack-clash-protection       Increased reliability of stack overflow detection
-fstack-protector              Overflow Erkennug via Canary (variants: all, strong) (RHEL6+)
-mcet -fcf-protection          Control flow integrity protection (future)
# Optimierung
-O2                            Recommended optimizations
-pipe                          Compile time optimization (avoid temporary files)
# Linker
-WL,-z,defs                    Detect and reject unterlinking
-Wl,-z,now                     Disable lazy binding (RHEL7+)
-Wl,-z,relro                   Read-only segments ofter relation (RHEL6+)
# Fehlerbehandlung
-fasynchronous-unwind-tables   Increased reliability of backtraces
-fexceptions                   Enable table-based thread cancellation
# Object Structure / Introspection
-fpic -shared                  No text relocations for shared libraries
-fplugin=annobin               Inquire about hardening options, ABI compatability
# Debugging Informationen
-g                             Add debuggin information and labels
-grecord-gcc-switches          Compilerflags Metadata als debugging info
# Warnungen und Hinweise
-Wall                          Recommended compiler warnings
-Werror=format-security        Reject potentially unsafe format strings
-Werror=implicit-function-declaration   Reject missing function prototypes
```

Siehe auch: `https://developers.redhat.com/blog/2018/03/21/compiler-and-linker-flags-gcc/`

```
1 void fun() {
2     char *buf = alloca(0x100);
3     /* Don't allow gcc to optimise away the buf */
4     asm volatile("" :: "m" (buf));
5 }
```

Siehe auch: https://idea.popcount.org/2013-08-15-fortify_source/

```
1  08048404 <fun>:
2  push   %ebp               ; prologue
3  mov    %esp,%ebp
4
5  sub    $0x128,%esp        ; reserve 0x128B on the stack
6  lea    0xf(%esp),%eax     ; eax = esp + 0xf
7  and    $0xfffffff0,%eax   ; align eax
8  mov    %eax,-0xc(%ebp)    ; save eax in the stack frame
9
10 leave                     ; epilogue
11 ret
```

Siehe auch: https://idea.popcount.org/2013-08-15-fortify_source/

```
08048464 <fun>:
push   %ebp                ; prologue
mov    %esp,%ebp

sub    $0x128,%esp         ; reserve 0x128B on the stack

mov    %gs:0x14,%eax       ; load stack canary using gs
mov    %eax,-0xc(%ebp)     ; save it in the stack frame
xor    %eax,%eax           ; clear the register

lea    0xf(%esp),%eax      ; eax = esp + 0xf
and    $0xfffffff0,%eax    ; align eax
mov    %eax,-0x10(%ebp)    ; save eax in the stack frame

mov    -0xc(%ebp),%eax     ; load canary
xor    %gs:0x14,%eax       ; compare against one in gs
je     8048493 <fun+0x2f>
call   8048340 <__stack_chk_fail@plt>

leave                      ; epilogue
ret
```

Siehe auch: https://idea.popcount.org/2013-08-15-fortify_source/

-D_FORTIFY_SOURCE=2

```
1 void fun(char *s) {
2     char buf[0x100];
3     strcpy(buf, s);  // Though you should prefer strncpy anyways! ;)
4     /* Don't allow gcc to optimise away the buf */
5     asm volatile("" :: "m" (buf));
6 }
```

Siehe auch: https://idea.popcount.org/2013-08-15-fortify_source/

-D_FORTIFY_SOURCE=2

```
08048450 <fun>:
push   %ebp                ; prologue
mov    %esp,%ebp

sub    $0x118,%esp         ; reserve 0x118B on the stack
mov    0x8(%ebp),%eax      ; load parameter  s  to eax
mov    %eax,0x4(%esp)      ; save parameter for strcpy
lea    -0x108(%ebp),%eax   ; count  buf  in eax
mov    %eax,(%esp)         ; save parameter for strcpy
call   8048320 <strcpy@plt>

leave                      ; epilogue
ret
```

Siehe auch: https://idea.popcount.org/2013-08-15-fortify_source/

```
1  08048470 <fun>:
2  push   %ebp                ; prologue
3  mov    %esp,%ebp
4
5  sub    $0x118,%esp         ; reserve 0x118B on the stack
6  movl   $0x100,0x8(%esp)    ; save value 0x100 as parameter
7  mov    0x8(%ebp),%eax      ; load parameter  s  to eax
8  mov    %eax,0x4(%esp)      ; save parameter for strcpy
9  lea    -0x108(%ebp),%eax   ; count  buf  in eax
10 mov    %eax,(%esp)         ; save parameter for strcpy
11 call   8048370 <__strcpy_chk@plt>
12
13 leave                       ; epilogue
14 ret
```

Siehe auch: https://idea.popcount.org/2013-08-15-fortify_source/

```c
/* Copyright (C) 1991–2018 Free Software Foundation, Inc.
This file is part of the GNU C Library.
The GNU C Library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.
The GNU C Library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.
You should have received a copy of the GNU Lesser General Public
License along with the GNU C Library; if not, see
<http://www.gnu.org/licenses/>.  */

#include <stddef.h>
#include <string.h>
#include <memcopy.h>

#undef strcpy

/* Copy SRC to DEST with checking of destination buffer overflow.  */
char * __strcpy_chk (char *dest, const char *src, size_t destlen)  {
    size_t len = strlen (src);
    if (len >= destlen)
        __chk_fail ();
    return memcpy (dest, src, len + 1);
}
```

Siehe auch: http://sourceware.org/git/?p=glibc.git;a=blob_plain;f=debug/strcpy_chk.c;hb=HEAD