Programmierung auf 64-Bit Plattformen

Michael Matz

matz@suse.de

SuSE Linux AG

Warum 64 Bit?

- grosser Addressraum
- Code für inherent 64 bittige Probleme (LFS)
- Ausnutzen neuer Features der Architektur

Warum 64 Bit?

- grosser Addressraum
- Code für inherent 64 bittige Probleme (LFS)
- Ausnutzen neuer Features der Architektur
- aber: Höherer Speicheroverhead

Eigenschaften von AMD64 (vs. x86)

- +8 int regs `%r8 - %r15` (weniger Spilling)
  Keine `%ah` mehr, dafür haben alle Register eine 8 Bit Variante
- +8 fp regs
- Hypertransport (schnelle NUMA Architektur)
- modernes ABI
- SSE2 für float/double (kein x87 Stack)
- `long` ist 64 Bit
- 32-Bit Registerschreibzugriffe löschen obere 32 Bit (Zero-Extend kostenlos)

Eigenschaften von AMD64 (vs. x86)

- +8 int regs `%r8 - %r15` (weniger Spilling)
  Keine `%ah` mehr, dafür haben alle Register eine 8 Bit Variante
- +8 fp regs
- Hypertransport (schnelle NUMA Architektur)
- modernes ABI
  - RIP Addressing (kein Zusatzregister für -fPIC notwendig)
  - register passing (schnelle Funktionsaufrufe)
  - schnelles syscall-Interface
  - CFI per default (backtraces gehen immer)
- SSE2 für float/double (kein x87 Stack)

32/64 Koexistenz

- korrekte Installation, Kernel- und Toolchain-Support notwendig
- 32Bit libs: */lib
- 64Bit libs: */lib64
- Includes sind gemeinsam

32/64 Entwicklung auf SL 9.0

- biarch-Compiler
  default: 64Bit Code wird erzeugt
  mit `-m32`: 32Bit Code wird erzeugt
- -L Argumente sind anzupassen (lib64!), -I nicht notwendig
- Installation von Libraries anpassen
- Debuggen mit gdb oder gdb32
- Personality umschaltbar mit linux32
- (Beispiel hello.c)

Stolperfallen

- Typgrössen / -alignment
- Int vs. long
- Prototypen
- Variable Argumente
- Misc

Typgrössen / -alignment

Typ	x86	AMD64
long	4	8
pointer	4	8
long double	12	16

$\longrightarrow$ Grösse und Alignment von structs können sich ändern.

Ausweg: `sizeof(type)`, `offsetof(struct, member)`

Int vs. long

- Pointer passen nicht in `int`'s (cf. `intptr_t`)
- Literale ohne Typangabe sind ints:
  - `long t = 1 < < a;` vs.
  - `long t = 1L < < a;`
- POSIX Typen benutzen!

Prototypen

- immer benutzen!
- ohne: Returntype `int`; Argumente unterliegen Promotion
- Bsp: `malloc()` ohne Prototyp (Header vergessen einzubinden) gibt 32 Bit zurück (i.e. keinen kompletten Pointer)

Variable Argumente

- Aufrufer und Aufgerufener müssen in Typ übereinstimmen
- 32-Bit Werte werden nicht auf 64 Bit erweitert (NULL != 0)
- Nie direkt auf va_list Variablen zugreifen, immer über Makros aus `<stdarg.h>`. Addressen darf man bilden.
  - `va_start, va_end, va_copy, va_arg`
  - `va_end` nicht vergessen!
- (Beispiel va1.c)

Portable (C99) Typen

- `<stdint.h>`:
  - Definierte Grösse: `intN_t` (N $\in$ 8,16,32,64 ...)
  - Mindestgrösse: `int_leastN_t`
  - Schnellster Typ mit mind. Grösse: `int_fastN_t`
  - Pointer als Integer: `intptr_t`
  - Grösster Inttyp: `intmax_t`
  - Für unsigned ”u” davor
- `<inttypes.h>`: Makros für printf/scanf zur Behandlung dieser Typen.
- Pointer: ”%p”; `size_t` Vars: ”%Z”
- (Beispiel maxint.c)

Verschiedenes

- signed char/unsigned char/char
- Auswertung von floating point Ausdrücken (float.c)
- Typkorrektheit (`extern long x;` in einer Datei, `int x;` in einer anderen)
- shared libs: -fPIC (-fpic) muss (!) benutzt werden (ld stellt dies sicher)
- Schleifenindex für Arrayzugriff: besser `size_t` als `int` (array.c)

- a