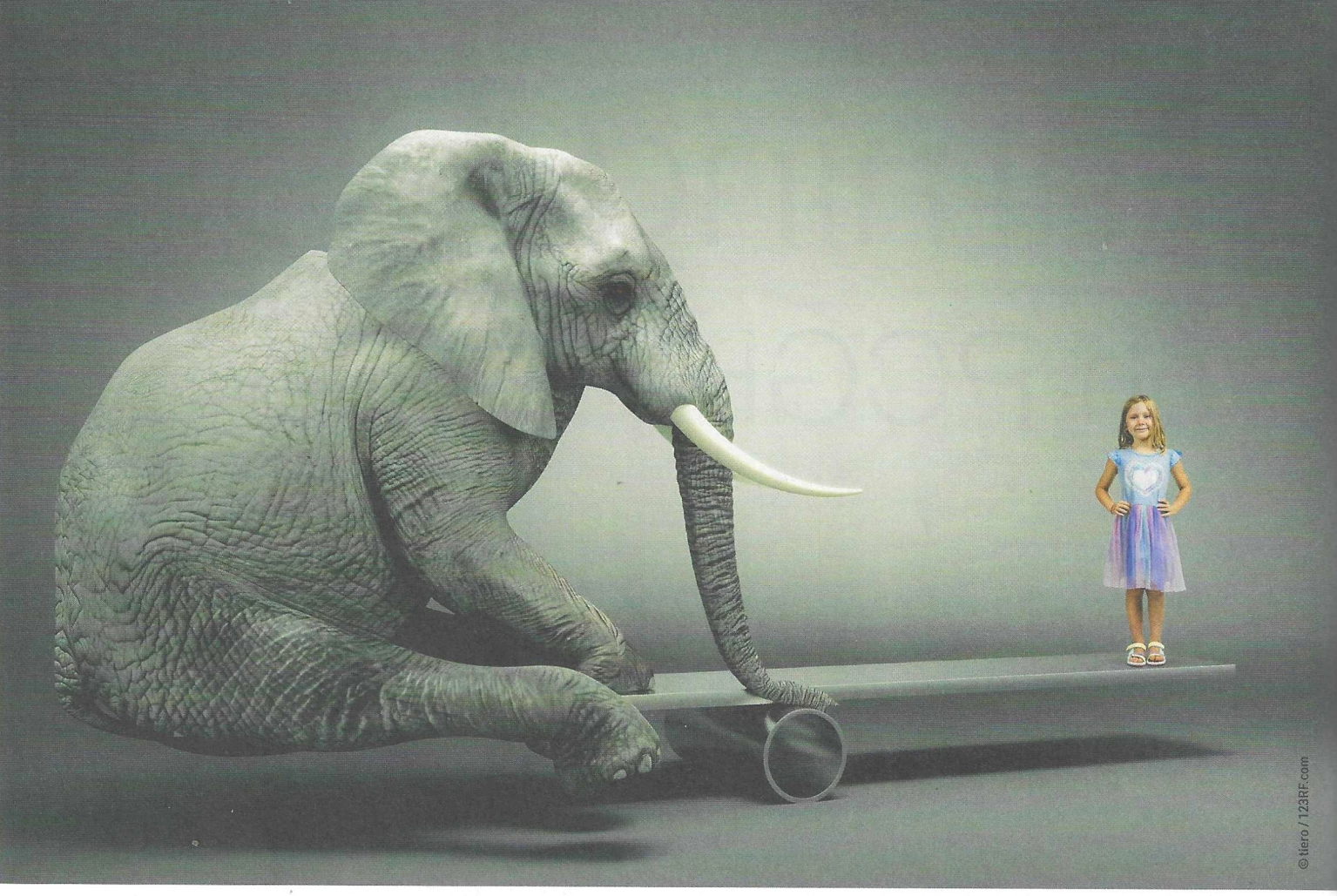


Kernel- und Treiberprogrammierung mit Linux – Folge 127

Klein, aber oho!

Im Quellcode des Kernels findet sich mit der Nolibc eine Bibliothek, die sich bei näherem Hinsehen als magischer Schlüssel zu System Calls und damit zu äußerst kompakten Systemen entpuppt. Eva-Katharina Kunst, Jürgen Quade

Der Autor

Eva-Katharina Kunst ist seit den Anfängen von Linux Fan von Open Source. **Jürgen Quade**, Professor an der Hochschule Niederrhein, gibt auch für Unternehmen Schulungen zu den Themen Treiberprogrammierung und Embedded Linux.

Machen oder machen lassen? Wer in seinen Programmcode eine Bibliotheksfunktion wie `printf()` oder `exit()` einfügt, lässt den Kernel antreten. Bei `printf()` schreibt nicht die Applikation die Daten, sondern beauftragt den Betriebssystemkern damit. Analog beendet `exit()` nicht das Programm, sondern erteilt nur den Auftrag dazu. Was im ersten Augenblick wie dozierende Erbsenzählerei aussieht, ist eine mehr als angenehme Arbeitsteilung. Es verdeutlicht, dass sich der Betriebssystemkern als ausgesprochen geduldiger Befehlsempfänger verdingt und Programme in weiten Teilen nur Dienste des Kernels in Anspruch nehmen. Solche Dienste nennt man im Kernel-Jargon System Calls.

System Calls entscheiden zentral über die Leistungsfähigkeit eines Betriebssystems. So bietet jedes ernst zu nehmende OS System Calls für Routinearbeiten wie die Ausgabe und das Einlesen von Daten, das Erzeugen und Beenden von Rechenprozessen oder das Lesen und Setzen der Uhrzeit. Die Liste der Linux-System-Calls ist lang und wächst mit jeder neuen Version. Linux 6.2 bietet auf einer 64-Bit-x86-Plattform beispielsweise rund 450 System Calls an.

Um noch einmal zu dozieren: Vielen Programmiererinnen und Programmierern entgeht der Unterschied zwischen einem System Call und einer Bibliotheksfunktion. Das liegt daran, dass Bibliotheken eine Abstraktionsschicht einbauen,

die jeden System Call transparent auf eine Funktion abbildet. Bei `printf()` handelt es sich beispielsweise um eine Bibliotheksfunktion, die auf dem System Call `write` basiert, und `exit()` setzt auf einem gleichnamigen System Call auf. Die Bibliotheksfunktion `syscall()` ermöglicht übrigens, in der Standardbibliothek noch nicht unterstützte System Calls aufzurufen, sofern man die entsprechende Parametrierung kennt.

Auslöser

Das System-Call-Interface legt fest, wie sich die Dienste des Kernels technisch anfordern lassen. Klassischerweise wird das über Software-Interrupts realisiert. Zum Auslösen eines Software-Interrupts bietet jede CPU-Architektur einen eigenen Maschinenbefehl. Bei x86-Prozessoren von Intel oder AMD lautete der Befehl ursprünglich beispielsweise `int` wie Interrupt. Der Befehl versetzt die CPU in den Interrupt-Modus. Sie rettet dann den aktuellen Zustand der CPU und arbeitet anschließend die zugehörige Interrupt-Service-Routine (ISR) ab. Die ISR als Teil des Betriebssystemkerns bearbeitet dann auf Basis der CPU-Register-Inhalte 1 den Auftrag.

Wie man über den Aufruf des System Calls `write` ein „Hello, World" in Assembler realisiert, zeigt Listing 1. Die gleichnamige Funktion `write()` der Standard-C-Bibliothek erwartet dieselben Parameter wie der System Call: den File Descriptor, der das Ziel für die Ausgabe bestimmt, die Adresse, an der die Daten zu finden sind, und die Anzahl der Bytes, die ausgegeben werden sollen 2.

Linus Torvalds hat vorgegeben, wie man dem Kernel die Daten des Systemaufrufs übergibt. Bei der x86-Architektur schreibt man den ersten Parameter (den File Descriptor) in das CPU-Register EBX. Der zweite Parameter, die Adresse der Daten, findet im CPU-Register ECX seinen Platz. Den dritten Parameter, die Anzahl der Bytes, legt man im Register EDX ab. Zum Belegen der CPU-Register mit den Werten dient der Assembler-Befehl `movl`. Die System Calls selbst sind durchnummeriert, wobei `write` die Nummer 4 hat, die man in das Register EAX schreibt 3. Listing 1 übergibt dem Kernel mit dem System Call `exit` (Syscall-Nummer 1) noch einen zweiten Auftrag.

Aus dem in `hello-int.s` gespeicherten Assembler-Code generieren Sie per `cc -no-pie -Os -static hello-int.s -o hello-int` das Binary `hello-int`, das nach dem Aufruf per `./hello-int` den berühmten Gruß ausgibt.

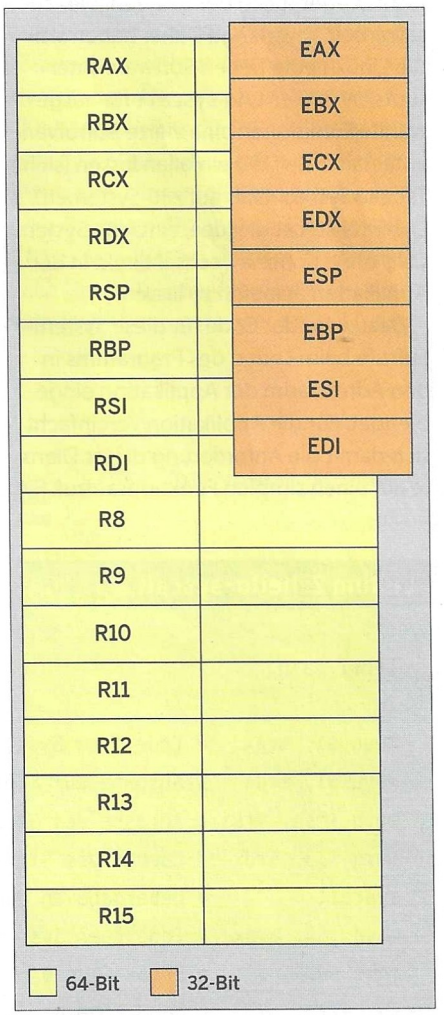
Modern befehlen

Die Methode, einen System Call über einen Software-Interrupt auszulösen, gilt heute als antiquiert. Die Hersteller der x86-Architektur haben mit `sysenter` (Intel) und `syscall` (AMD) schnellere, auf die Aufgabe hin optimierte Maschinenbefehle in ihre CPUs eingebaut. Insbesondere `syscall` kommt dabei auf 64-Bit-Systemen als Standardmechanismus zum Einsatz.

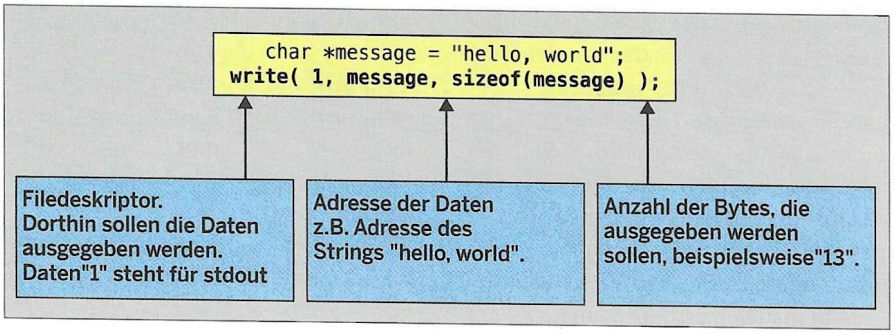
Eine der Optimierungen besteht darin, die Rücksprungadresse, also die Adresse, an der das Programm nach Ausführung des System Calls weiterarbeiten soll, im CPU-Register RCX abzulegen. So muss man sie nicht langwierig auf den Stack retten und später von dort restaurieren. Außerdem hat Torvalds damit die Übergabe der System-Call-Parameter der Übergabekonvention von Funktionsaufrufen angenähert. Das erspart der C-Bibliothek eine Menge Umkopieren. Bei dieser Gelegenheit wurden auch gleich die System-Call-Nummern neu vergeben.

Listing 1: hello-int.s

```
.text
.globl main
main:
  movl $4,%eax       # Code fuer System Call "write" (32 Bit)
  movl $1,%ebx       # File Descriptor fd (1=Stdout)
  movl $message,%ecx # Adresse des Texts ("message")
  movl $13,%edx      # Laenge des auszugebenden Textes
  int $0x80          # Software-Interrupt, Auftrag an das OS
  movl $1,%eax       # Code fuer System Call "exit"
  movl $0,%ebx       # Error Code 0
  int $0x80          # Software-Interrupt, Auftrag an das OS
  ret
.data
message:
  .ascii "hello, world\n"
```



1 **Die CPU-Register der 32- und 64-Bit-x86-Architektur.**



2 `write()` **ist eine Funktion der Standard-C-Bibliothek.**

Listing 2 enthält den Assembler-Code für unser Hello-World-Beispiel. Ein `cc -no-pie -Os -static hello-syscall.s -o hello-syscall` generiert wieder das Grußprogramm, dessen Ausführung zeigt den Erfolg.

Jeder Aufruf eines System Calls führt zu einem Kontextwechsel, der (etwas vereinfacht ausgedrückt) der CPU sehr viel Zusatzarbeit abverlangt. Daher haben sich die Linux-Leute neben Software-Interrupts, `sysenter` und `syscall` für ausgewählte Funktionen eine vierte Aufrufvariante namens vDSO einfallen lassen (siehe Tabelle System Calls auf x86-Systemen). Dahinter steckt die Idee, einfache System Calls ohne Kontextwechsel direkt in der Applikation ablaufen zu lassen.

Dazu wird der Code für diese Systemaufrufe beim Laden des Programms in den Adressraum der Applikation eingeblendet. Für die Applikation vereinfacht sich damit die Anforderung dieser Dienste auf einen simplen Funktionsaufruf. Ein Blick in den Linux-Quellcode offenbart allerdings, dass sich bisher nur die System Calls `clock_gettime`, `gettimeofday`, `getcpu`, `time` und `clock_getres` auf diese Art erreichen lassen. Zudem erschwert eine ausgefeilte Sicherheitstechnik (zufällige Adresslage von Funktionen durch Address Space Layout Randomization, kurz ASLR) das Auffinden der Funktionsadressen. Dankenswerterweise schlägt sich die Standard-C-Bibliothek mit diesen Spitzfindigkeiten herum und entlastet so die Entwickler.

13 zu 3

Über das Kommandozeilenwerkzeug Strace lassen sich übrigens die von einem Stück Software verwendeten System Calls sehr schön nachverfolgen. Abbildung 4 demonstriert das für den Aufruf des Hello-World-Programms aus Listing 3. Strace zählt hier bei einem statisch gebundenen Binary überraschenderweise 13 System Calls, wo man eigentlich genau drei – `execve`, `write` und `exit` – erwarten würde. Offensichtlich bauen Compiler und Linker hier einigen Overhead ein.

Auf dem Massenspeicher belegt das statisch gebundene und hinsichtlich der Größe optimierte Programm deutlich über 850 KByte. Übersetzt man das Assembler-Programm aus Listing 2 mit denselben Compiler-Optionen wie das C-Programm, ergibt sich allerdings derselbe Memory-Footprint (und eine beinahe identische Anzahl an System Calls).

Ohne Bibliothek

Tatsächlich bindet der Linker standardmäßig einen Startup-Code sowie die C-Bibliothek zum eigentlichen Programmcode. Das erklärt zwar den Speicherhunger, ist aber im Fall des Assembler-Programms definitiv unnötig. Um das zu verhindern, geben Sie dem Compiler die Optionen `-nostdlib` und `-nolibc` mit. Er reicht sie an den automatisiert aufgerufenen Linker durch.

Dass jedes C-Programm mit der Funktion `main()` beginnt, legt übrigens der Startup-Code fest. Ohne Startup-Code heißt der Einsprungpunkt `_start`, und folglich kann man im Assembler-Quellcode das `main` gegen `_start` austauschen. Mit dieser Änderung und den angegebenen Optionen begnügt sich das Hello-World-Programm mit 9000 Bytes. Unter der Obhut von Strace ausgeführt, bleiben schließlich nur noch die drei nachvollziehbaren System Calls übrig 5.

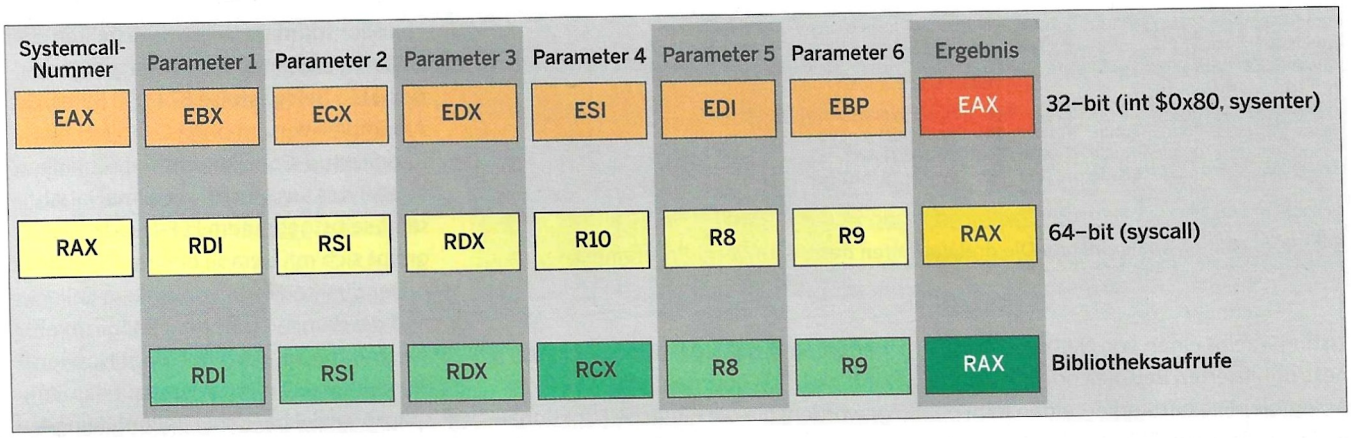
Der zweite (`write`) und dritte (`exit`) System Call entsprechen dem programmierten Code. Der erste (`execve`) flutet das Codesegment mit unserem Code und gehört damit noch zum Startmechanismus eines Programms. Dabei wird durch

Listing 2: hello-syscall.s

```
.text
.global main
main:
  movq $1, %rax   # Code fuer System Call "write" (64 Bit)
  movq $1, %rdi   # Ausgabe auf Stdout
  movq $msg, %rsi # Adresse des Strings
  movq $13, %rdx  # Laenge des Strings
  syscall         # Uebergabe an den Kernel
  movq $60, %rax  # Code fuer System Call "exit" (64 Bit)
  movq $0, %rdi   # Error Code 0
  syscall         # Uebergabe an den Kernel
.data
msg: .ascii "Hello, World\n"
```

Listing 3: Hello, World!

```
#include <stdio.h>
int main( int argc, char **argv,
char **envp )
{
  printf("Hello, World!\n");
  return 0;
}
```



3 **Die Parametrierung von System Calls und Bibliotheksfunktionen erfolgt über CPU-Register.**

`fork()`, `pthread_create()` oder `clone()` getriggert, zunächst eine exakte Kopie des aktiven Jobs (im konkreten Fall der Shell) in Auftrag gegeben. Diese Dublette gerät dann durch Überschreiben des Codesegments, wie hier zu sehen, zum gewünschten Programm (hier `syscall`).

Schrumpfkur

9000 Bytes scheinen allerdings für ein paar Zeilen Assembler nach wie vor etwas übertrieben. Tatsächlich sorgt ein umfangreicher ELF-Header für die Größeninflation. ELF steht für das Executable and Linking Format, das definiert, wie notwendige Metainformationen mit dem eigentlichen Code und den Daten in einer (ausführbaren) Datei abgelegt werden. Schließlich bestehen Programme aus unterscheidbaren Komponenten, den Segmenten (Code, Daten, Stack, etc.). Außerdem steht dort, für welche Plattform der Code bestimmt ist und ob es Debug-Infos gibt. Beim Laden eines Programms interpretiert der Kernel den Header und kann die in der Datei vorhandenen Daten passend im Hauptspeicher platzieren.

System Calls auf x86-Systemen

Methode	Beschreibung
`int`	Software-Interrupt (x86, 32 Bit)
`sysenter`	spezieller Maschinenbefehl (x86, 32 Bit)
`syscall`	spezieller Maschinenbefehl (x86, 64 Bit)
vDSO	Code in den Adressraum einblenden

Geht doch!

Tatsächlich erzeugen die Unix-Werkzeuge einen komplexen Header, der auf dem internen Speicher weitaus mehr Platz belegt als unbedingt notwendig. Zwar ist es nicht ganz trivial, eine minimale ELF-Datei zu erzeugen, aber eben auch nicht unmöglich.

Als Erstes benötigen Sie dazu ein Werkzeug, das nicht zwangsweise einen ELF-Header respektive ein ELF-Format erzeugt. Dazu bietet sich Nasm an, der freie Netwide Assembler, der auf so ziemlich jeder Plattform x86-Maschinencode generieren kann. Die Assembler-Syntax von Nasm ist allerdings invers zur Intel-Syntax, bei der man zuerst die Quelle (Quellregister oder -daten) und dann das Ziel (Zielregister) angibt. Nasm erwartet stattdessen zuerst das Ziel und dann die Quelle respektive die Daten. Der Assembler-Befehl `mov edi, eax` bedeutet in Nasm also, dass es gilt, den Inhalt des CPU-Registers EAX in das CPU-Register EDI zu verschieben (faktisch: zu kopieren). Unter Ubuntu installieren Sie Nasm mit dem Kommando `apt install nasm`.

```
cc -Wall -Os  -static  hello.c   -o hello
quade@ezs-cocka:~/syscall$ strace ./hello
execve("./hello", ["./hello"], 0x7ffe69110870 /* 49 vars */) = 0
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffc3269dfa0) = -1 EINVAL (Das Argument ist ungültig)
brk(NULL)                                = 0x902000
brk(0x9031c0)                            = 0x9031c0
arch_prctl(ARCH_SET_FS, 0x902880)        = 0
uname({sysname="Linux", nodename="ezs-cocka", ...}) = 0
readlink("/proc/self/exe", "/home/quade/syscall/hello", 4096) = 25
brk(0x9241c0)                            = 0x9241c0
brk(0x925000)                            = 0x925000
mprotect(0x4bd000, 12288, PROT_READ)     = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0xa), ...}) = 0
write(1, "Hello, world\n", 13Hello, world
)           = 13
exit_group(0)                            = ?
+++ exited with 0 +++
```

4 **Strace verfolgt die System Calls des Hello-World-Programms auf der Konsole.**

```
cc  -static -nostdlib -nolibc  syscall.s   -o syscall
quade@ezs-cocka:~/syscall$ strace ./syscall
execve("./syscall", ["./syscall"], 0x7ffd5240e970 /* 49 vars */) = 0
write(1, "hello, world\n", 13hello, world
)           = 13
exit(0)                                  = ?
+++ exited with 0 +++
```

5 **Die schlanke Assembler-Variante: „Hello, World" in drei System Calls.**

```
quade@ezs-cocka:~/syscall$ nasm -f bin -o hello-tiny hello-tiny.s
quade@ezs-cocka:~/syscall$ chmod +x hello-tiny
quade@ezs-cocka:~/syscall$ ./hello-tiny
Hello, world!
quade@ezs-cocka:~/syscall$ ls -lrt hello syscall hello-tiny
-rwxrwxr-x 1 quade quade 871896 Feb 24 20:19 hello
-rwxrwxr-x 1 quade quade   9032 Feb 24 20:27 syscall
-rwxrwxr-x 1 quade quade    157 Feb 24 20:28 hello-tiny
```

6 Klein, kleiner, am kleinsten: Die drei Varianten des Hello-World-Programms.

Listing 4 zeigt einen von Nathan Otterness publizierten Beispielcode, der im Assembler-File hartkodiert einen minimalen, passenden ELF-Header erzeugt, indem er Byte für Byte die Daten zusammenstöpselt. Damit übernimmt der Assembler die Aufgabe des Linkers, und es genügt, die vorliegende Datei zu assemblieren und in Maschinencode zu übertragen. Und siehe da: Aus 9000 Bytes sind plötzlich übersichtliche 157 Bytes geworden, die die einfache Funktionalität passend repräsentieren.

In Abbildung 6 sehen Sie, wie aus dem Assembler-Code in `hello-tiny.s` das ausführbare Programm `hello-tiny` entsteht und ausgeführt wird. Darüber hinaus zeigt der Screenshot den Speicher-Footprint der hier vorgestellten drei Hello-Versionen. Das statisch gebundene `hello` belegt stolze 871 896 Bytes, die Assembler-Version ohne C-Bibliothek und Startup-Code kommt mit schlanken 9032 Bytes aus, und die Minimalvariante samt selbstgebautem ELF-Header begnügt sich mit gerade einmal 157 Bytes.

Der direkte Aufruf von System Calls und die damit verbundene Möglichkeit, Programme auf das unbedingt Notwendige zu schrumpfen, deutet auf das Einsparpotenzial bei der Erstellung (eingebetteter) Systeme hin. Mit Linux als skalierbarem Betriebssystemkern (siehe Kasten Kernel eindampfen) und mit einem ohne Nutzung der Standardbibliothek gebauten Userland fällt das Gesamtsystem erfrischend kompakt aus.

Listing 4: hello-tiny.s

```
[bits 64]
file_load_va: equ 4096 * 40
db 0x7f, 'E', 'L', 'F'
db 2
db 1
db 1
db 0
dq 0
dw 2
dw 0x3e
dd 1
dq entry_point + file_load_va
dq program_headers_start
dq 0
dd 0
dw 64
dw 0x38
dw 1
dw 0x40
dw 0
dw 0
program_headers_start:
dd 1
dd 5
dq 0
dq file_load_va
dq file_load_va
dq file_end
dq file_end
dq 0x200000
entry_point:
  ; Set eax (and, by extension, rax) to 1. (The write syscall number)
  xor eax, eax
  inc eax
  ; Set edi (and, by extension, rdi) to 1. (The stdout file descriptor)
  mov edi, eax
  ; Set esi (and, by extension, rsi) to the string's virtual address.
  mov esi, file_load_va + message
  ; Set edx (and, by extension, rdx) to the string's length.
  xor edx, edx
  mov dl, message_length
  ; Issue the write syscall
  syscall
  ; rax is already 0 assuming the write succeeded, so set the next syscall
  ; number to 60, for the exit syscall.
  mov al, 60
  ; Set the exit status to 0.
  xor edi, edi
  ; Exit the program.
  syscall
code_end:
message: db `Hello, World!\n`
message_length: equ $ - message
file_end:
```

Von diesen Vorzügen angespornt hat sich der Entwickler Willy Tarreau darangemacht, Code zu schreiben, der wesentliche Funktionen der Standard-C-Library implementiert, aber eben nicht als Bibliothek zum Programm gebunden wird. Das Ergebnis abstrahiert die Funktionalitäten über Makros und statische Funktionen. Diese rufen wie hier gezeigt ohne Umschweife die System Calls auf. Weil es sich bei seinem Code eben um keine normale C-Bibliothek handelt, lag es 2017 nahe, das Projekt Nolibc zu taufen.

Seltsamer Name

Obwohl es sich um Userland-Code handelt, findet sich Nolibc überraschenderweise im Linux-Quellcode wieder. Das allein ist schon bemerkenswert, denn für den Kernel selbst spielt der Code keine Rolle, wohl aber für automatisierte Tests des Betriebssystemkerns.

Genau zu diesem Zweck hat Kernel-Entwickler Paul E. McKenney die Softwarekomponente integriert. Er baut damit besonders schlanke Testsysteme zusammen, die insbesondere dazu dienen, RCU-Implementierungen (Read Copy Update) im Kernel zu quälen. Nolibc beschleunigt das Generieren der Systeme und durch die reduzierte Größe auch das Booten in einer emulierten oder simulierten Umgebung. Das spart nicht nur Ressourcen: Im Embedded-Bereich erhöht die Reduktion der Codemenge darüber hinaus die Sicherheit.

Beispiel		Struktur der Funktionsnamen
write(1, "Hello, world!\n", 13);	Libc-Interface	<name>()
sys_write(1, "Hello, world!\n", 13);	architekturunabhängig	sys_<name>()
syscall(4,1, "Hello, world!\n", 13);	architekturspezifisch	my_syscallX

7 **Die Softwarearchitektur der Nolibc.**

No-Go im Kernel

Der Code der Nolibc befindet sich komplett in den Header-Dateien des Linux-Quellcodes unter `/usr/src/linux/tools/include/nolibc/`. Code in Header-Dateien? Unter C-Profis gilt das normalerweise als absolutes No-Go.

Tarreau bevorzugt diese Architektur aber wegen der wesentlich vereinfachten Anwendung, die kein Vorkompilieren oder gesondertes Zusammenbinden von diversen Quellcodedateien erfordert. Es genügt, beim Kompilieren den Pfad zu den Nolibc-Headern anzugeben, um zu einem Executable zu kommen. Ein kleiner Nachteil dieses Ansatzes: Viele Funktionen der schlanken C-Nicht-Bibliothek landen auch dann im Programm, wenn sie nicht benötigt werden. Umfasst eine Software mehrere Quellcodedateien, bindet der Linker auch die Standardfunktionen – weil in den Nolibc-Headern statisch definiert – mehrfach dazu.

In Kombination mit dem üppigen ELF-Header geraten die Programme damit nicht so hübsch klein wie bei Assembler-Code und selbstgestricktem Header, jedoch signifikant kleiner als auf einem Standardsystem. Dem rund 850 KByte großen, klassisch generierten Hello-World-Programm stehen gerade einmal 9 KByte in der Nolibc-Variante gegenüber.

Befindet sich der Linux-Quellcode im Verzeichnis `/usr/src/linux/`, erzeugt das Kommando in Listing 5 aus dem Hello-World-Quellcode in Listing 3 ein kompaktes `hello-nolibc`.

Besserwisser

Die Einsatzmöglichkeiten der Nolibc beschränken sich keineswegs auf die x86-Architektur. Vielmehr ist die Nicht-Bibliothek wie der Kernel selbst hochportabel und auf die unterschiedlichen Architekturen wie ARM oder RISC-V angepasst. Über Makros bindet sie dazu den zur Architektur gehörenden und in Header-Dateien befindlichen Assembler-Code zum Aufruf der System Calls ein 7. Die Nicht-Bibliothek im Quellcode des Kernels verspricht damit ein besonderes softwaretechnisches Schmankerl und macht am Ende alles klein. Small ist nicht nur beautiful, sondern auch nachhaltig. (*jlu*)

Listing 5: Kompaktes „Hello, World!"

```
$ cc -I /usr/src/linux/tools/include/nolibc -nostdlib -nolibc -no-pie \
    -fno-asynchronous-unwind-tables -Os hello.c -o hello-nolibc
```

Kernel eindampfen

Der Kernel ist extrem skalierbar. Etwas Zeit vorausgesetzt, lässt er sich beispielsweise exakt auf eine Hardware und die dort zu erfüllende Arbeit hin konfigurieren. Kein unnötiger Gerätetreiber und kein ungenutztes Subsystem belasten dann die Hardware. Um die Möglichkeiten zu demonstrieren, hat Linus Torvalds in das Kernel-Build-System das Target `make tinyconfig` eingebaut, das den kleinstmöglichen Kernel generiert. Für ein x86_64-System hat er schlanke 488 KByte und lässt sich auf einem leidlich modernen Notebook in weniger als einer Minute generieren. Zum Vergleich: Der Ubuntu-Kernel benötigt mehr als 11 MByte, die üppigen Gerätetreiber noch nicht einmal eingerechnet. Die Generierung dauert auf einem Multicore-System Stunden; kommt ein Single-Core zum Einsatz, schlafen Sie am besten eine Nacht darüber.

Dateien zum Artikel herunterladen unter
www.lm-online.de/dl/48904

Weitere Infos und interessante Links
www.lm-online.de/qr/48904