diet libc

Felix von Leitner
felix-dietlibc@fefe.de

März 2002

Agenda

1. Worum geht es?
2. Warum machst du so was?
3. Was kann es denn schon?
4. Die Zukunft

Worum geht es?

- Standard C Laufzeitumgebung
- So klein wie möglich, so schnell wie nötig
- Probleme lösen, weil sie *dringend* sind, nicht *der Vollständigkeit halber*!
- Referenz: Single Unix Specification 2, nicht glibc
- „besser als glibc“ sein
- Läuft auf x86, ARM, SPARC, Alpha, PowerPC, MIPS, HPPA, S/390

Warum machst du so was?

- *weil die glibc stinkt!* ;-)
- Kleine Software ist häufig überlegen (thttpd, djbware)
- Vorläufer-Projekte: libdjb, libowfat (ersetzt weite Teile der libc)
- grundlegendes Sicherheitsprinzip: Separation
- glibc macht das *viele kleine Programme* Modell fürchterlich ineffizient
- Schreibe eine libc mit nur dem nötigsten Code

Bist du bekifft? Warum hast du nicht einfach glibc aufgehackt?

„Wäre das nicht viel einfacher gewesen?“

- glibc ist total im Eimer (`popen`, `inet_ntoa`, `inet_aton`, ...)
- Weniger Hack-Value
- Ich habe dabei viel über Unix und kleine Software gelernt
- Ulrich Drepper ist manchmal ein bißchen schwierig in der Zusammenarbeit
- Außerdem stünde ich hier heute nicht ;-)

Kurz: glibc aufhacken vereint alle Nachteile mit keinem der Vorteile.

Warum portierst du die diet libc nicht auf *BSD?

- Ich kann BSD nicht leiden.
- Die BSD Leute sind für 3/4 der kaputten APIs verantwortlich, die Unix heute plagen. Der Rest kann Sun zugeschrieben werden.
- Die BSD Leute würden es nicht akzeptieren ohne $LICENSE==BSD
  (im Moment: GPL)
- Microsoft könnte Windoze mit meinem Code aufwerten
  (dazu kann ich es natürlich nicht kommen lassen).
- Rein technisch steht dem nichts im Wege, der Code ist mit kleinen Ausnahmen wie `openpty` portabel.

Warum GPL und nicht LGPL oder BSD?

Weil ich nicht verarscht werden will.

Jeder und sein kleiner Bruder machen gerade embedded Kram. Sogar Microsoft versucht sich mit *Embedded NT* (muhahaha) in dem Markt zu etablieren. Nur drei oder vier Firmen haben der Gemeinschaft irgend etwas nützliches zurückgegeben, die anderen verkaufen anderer Leute Code.

Wenn jemand eine kommerzielle Anwendung gegen die diet libc linken will, kann gerne mit mir über eine kommerzielle Lizenz verhandeln.

Effizienzmessung: Prozessorzyklen

Wir zählen die Zyklen, um `main() { return 0; }` auszuführen.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main(int argc,char* argv[]) {
  long a,b;
  asm("rdtsc":"=a"(a)::"edx");
  if (!fork()) { execve(argv[1],argv+1,environ); exit(1); }
  wait(0);
  asm("rdtsc":"=a"(b)::"edx");
  printf("%lu cycles\n",b-a);
}
```

Effizienzmessung: Speicherverbrauch

```
#include <stdio.h>

main() {
  FILE* f=fopen("/proc/self/statm","r");
  int size,resident,shared;
  if (fscanf(f,"%d %d %d",&size,&resident,&shared)==3)
    printf("%dK total size, %dK resident, %dK shared\n",
           size*4,resident*4,shared*4);
}
```

glibc vs. diet libc

	glibc	static glibc	diet libc
execve (Megazyklen)	1,340	0,330	0,215
Binary-Größe	4.751	432.883	1.528
Binary-Größe (gestrippt)	2.516	352.712	532
Speicher (shared)	296k	88k	12k
Speicher (non-shared)	52k	24k	20k

`ls -l` von 8 Dateien braucht mit der diet libc 1.160 Mega-Zyklen (inklusive Framebuffer-Ausgabe auf vesafb!).

Von den 532 Bytes sind 110 Code (Der Rest: ELF-Header und Alignment).

Speicher wird auf x86 in 4k-Pages alloziert. 8k für Stack und `.bss` sind für normale Programme unvermeidlich (`environ`, `errno`).

Spielt Effizienz bei 2 GHz Desktop CPUs noch eine Rolle?

Aber natürlich! Ich habe einen kleinen httpd namens *fnord* geschrieben. fnord wird von tcpserver (ähnlich inetd) gestartet.

```
for i in ‘seq 0 1000‘; do
  fget http://127.0.0.1:8000/index.html > /dev/null
done
```

Konfiguration	Ausgabe von time(1)
beides diet	0.14s user 0.54s system 63% cpu 1.076 total
glibc fget	0.83s user 0.79s system 76% cpu 2.125 total
beides glibc	0.73s user 0.95s system 53% cpu 3.114 total

Das sind ungefähr 2000 Hits pro Sekunde oder 166.3 Millionen Hits pro Tag (auf meiner single-CPU 1.333 MHz Athlon Desktop Kiste)!

Warum fget in den Benchmarks anfassen?

CGI! Viele Webserver sind für statischen Inhalt optimiert, aber für CGI müssen sie pro Anfrage fork() und execve() machen!

Also haben die Leute Scriptsprachen in Webserver eingebaut oder als Plug-In gebastelt und die Performance mit multi-threading zu tunen versucht. Beide Ansätze opfern Sicherheit und Stabilität für Performance.

Mit *fnord* auf einem voll dynamischen Webserver mit CGI kann man dank der diet libc immer noch 80 Millionen Hits pro Tag abarbeiten!

Design-Prinzipien

- Guter Code ist wichtiger als Programmiererzeit.
- Nicht kopieren, neu machen. Debuggen ist einfacher als Bloat entfernen.
- Jedes Byte zählt.
- Niemals etwas *„der Vollständigkeit halber“* implementieren.
- Abhängigkeiten mit allen Tricks verhindern.
- Multi-Platform Freundlichkeit (statische Header).
- Ruhig mal plattformspezifische Optimierungen machen, wenn es sich lohnt.

Was kann die diet libc denn schon?

- alle (?) System Calls; opendir, readdir, closedir; POSIX signals
- gmtime, localtime (with /etc/localtime), mktime, ctime, asctime
- stdio inklusive *printf und *scanf
- libpthread
- libdl und ld.so (bisher nur x86 und ARM, Alpha-Qualität)
- libm (bisher nur x86)

- malloc, realloc, free
- librpc (Mann war^H^H^Hist das eklig!)
- get(mnt,proto,serv,net,ut,gr,pw,sp)ent, syslog
- partial libresolv (dn_expand, res_mkquery, res_query...)
- gethostby(name,addr)/2, getaddrinfo, getnameinfo
- crypt mit md5crypt
- regcomp, regexec!
- large file support, 32-bit UID support

- qsort und bsearch
- networking glue: ntoh[sl], hton[sl], if_indextoname, if_nametoindex
- ctype (isalpha, isalnum, tolower) für ASCII und latin1
- iconv (bisher nur UTF8 <-> latin1)
- fnmatch, glob, ftw (noch nicht perfekt)
- rand und *rand48
- mk[sd]temp, openpty, getopt, getopt_long, getopt_long_only

Ist ja gut, reicht! ;-)

Was fehlt noch?

- ein reicher und großzügiger Sponsor
- regerror
- Ports auf IA64, 68k und Super-H (hehe)
- Locale (das bleibt auch so, Locale stinkt)
- ein bißchen hier, ein bißchen da

Super-H wird demnächst gemacht, und sei es nur weil uClibc es unterstützt.

Minimale interne Abhängigkeiten

- getservent und Konsorten benutzen nicht fopen
- atexit-Handling wird nur eingelinkt, wenn atexit benutzt wird
- printf referenziert nicht stdio (war gar nicht so einfach)
- Gleitkommazahlen-Unterstützung für printf kann wegkonfiguriert werden

Hilf Programmierern, kleinen Code zu schreiben

Wir benutzen Linker-Warnungen, wenn:

1. `assert` benutzt wurde (Debug-Code)
2. stdio benutzt wird (bloat)
3. `sprintf` benutzt wird (snprintf ist besser)
4. *printf/*scanf benutzt werden (Bloat, lieber -lowfat)
5. `system` benutzt wird (Sicherheitsproblem)
6. `mktemp`/`tempnam`/`tmpnam` benutzt werden (Race Condition)

Wie man die diet libc benutzt

Installation:

```
$ make
[...]
$ sudo make install
```

Benutzung:

```
$ diet gcc -o program main.c
$ CC="diet gcc -nostdinc" ./configure --prefix=/opt/diet
```

Cross-Compiler Unterstützung

- Weder Header noch Code dynamisch generiert
- Keine Abhängigkeiten von externen Headern
- Dieselben Header-Dateien für alle Plattformen (kein *bits/* wie bei glibc)
- Hängt nicht von Kernel-Headern ab (kein *linux/errno.h* wie glibc)
- Einfache Integration: `diet arm-linux-gcc -o program main.c`

Welche Programme funktionieren denn schon?

Erstaunlich viele (neben meinen eigenen natürlich).

sysvinit, net-tools (ifconfig), util-linux (mount, fdisk), gzip, bzip2, wget, syslogd, busybox, boa, ash, thttpd, wireless-tools, hdparm, gpm, mtr, lame, mpg123, portmap, openssl, openssh, ncurses, zsh, gawk, less, webfsd, links (!), vim (!!), mutt (hat regex Probleme), apache, und noch eine Menge mehr.

Ein paar Programme wurden sogar extra für die diet libc entwickelt: embutils, mininet, minit, fgetty, fget und fnord (von mir), diethotplug (wird wohl Teil von Kernel 2.5), ps und syslog (von Olaf Dreesen). Aber zuerst: -lowfat.

libowfat

(Es schreibt sich l-i-b-o-w-f-a-t, aber es wird „-lowfat“ ausgesprochen ;-))

libowfat reimplementiert die internen Hilfsroutinen von Dan Bernsteins Software als GPL. Beispiel:

```
#include <buffer.h>
int main() {
/* equivalent to: printf("%d\n",23); */
  buffer_putulong(buffer_1,23);
  buffer_putsflush(buffer_1,"\n");
}
```

printf: 5.8k, -lowfat: 1.2k

embutils

Meine Antwort auf busybox.

Implementiert: arch, basename, cat, chmod, chown/chgrp, chroot, chvt, clear, cp, dd, df, dirname, dmesg, domainname, echo, env, false, head, hostname, id, install, kill, ln, ls, md5sum, mesg, mkdir, mknod, mv, pwd, rm, rmdir, sleep, sync, tar, tee, touch, tr, true, tty, uname, uniq, wc, which, whoami, write, yes.

Kommt auch mit SOS-Versionen von cp, ln, ln -s, mv, rm.

mininet

Kleine Implementationen von ping und host. Geplant: ifconfig, route, arp, ...

minit

Neues init. Kleiner und mächtiger als sysvinit.

Features:

1. Abhängigkeiten
2. Kann dynamisch Dienste starten und beenden
3. Benutzt nicht /proc, System V IPC, IP oder Unix Domain Sockets
4. Läuft auch komplett auf einem read-only Dateisystem
5. Trotzdem ist /sbin/minit nur 7k groß

fgetty

Features:

1. Benutzt checkpassword (noch eine Erfindung von Dan Bernstein)
2. Dadurch ist die Authentisierung ohne Rekompilieren austauschbar
3. Kommt mit einem checkpassword mit /etc/shadow und MD5 Support
4. Trotzdem ist /sbin/fgetty nur 7k groß

fget (http/ftp Downloader)

Features:

1. Eigentlich keine. Kann eine einzelne URL saugen und auf stdout ausgeben.
2. HTTP und FTP
3. IPv6 und IPv4
4. `http_proxy` und `ftp_proxy` oder ohne Proxy

fnord (web server)

Features:

1. Wird von tcpserver gestartet
2. Statische Dateien (CGI kann reinkompiliert werden)
3. Kann URLs umschreiben für gzip Kompression
4. Virtuelle Hosts
5. Erstaunlich effizient

6. Dazuschaltbar: Index-Generierung (3k)

7. Trotzdem nur 13k (18k mit CGI) groß

Was noch gemacht werden muß

- Dynamischer Linker und libm für alle Plattformen
- Eine Test-Suite
- Mehr Programme angucken, z.B. X, kaffe...
- Mehr diet-Versionen von bestehenden Programmen schreiben
- dietlibc++?
- ein diet Java Runtime?

Unsere Tricks: Unified Syscalls

Den tatsächlichen Syscall-Code wiederverwenden. Jeder Syscall schreibt nur die Syscall-Nummer in das Register und springt zum gemeinsamen Code, der die Parameter in die Register kopiert, in den Kernel springt, und im Fehlerfall `errno` setzt. Das reduziert jeden Syscall auf ein paar Bytes.

Unsere Tricks: weak ELF symbols

ELF definiert „*weak symbols*“. Wenn man ein Symbol referenziert, das in dem bereits gelinkten Code bereits weak exportiert wird, dann sucht der Linker nicht mehr in den anderen Libraries und Objektdateien nach dem Symbol. Wenn ein Symbol weak und normal in einer Library ist, wird ansonsten das normale bevorzugt.

Was ist mit uClibc?

uClibc ist ein älteres Projekt als die diet libc.

Es ist in der gleichen Effizienzklasse wie die diet libc (d.h. viel besser als glibc) und unterstützt Super-H. In allen anderen Disziplinen erzeugt die diet libc kleineren Code bei gleicher, vollständigerer oder schnellerer Implementation.

Was ich bei uClibc nicht mag ist daß sie so viel Code kopieren. Das ist einfacher, aber es führt zur Dunklen Seite der Macht. Wenn man Code kopiert, kann man nicht viel besser sein als die Implementation, bei der man sich bedient hat. Der meiste Code in der uClibc scheint von anderen Projekten kopiert worden zu sein (stdio, libm und der dynamische Loader z.B.).

Your mileage may vary ;-)

Unterschiede zur glibc

1. `openpty` funktioniert nur mit devptsfs
2. getopt und Konsorten sortieren argv nicht
3. Braucht `_BSD_SOURCE` für `caddr_t`, `ulong`, `u_long` etc.
4. Braucht `_GNU_SOURCE` für `u_int32_t` etc.
5. Die Fake-Locale, natürlich
6. regcomp kann keine nicht-extended regular expressions

7. Die Ausgabe von Fließkommazahlen ist ziemlich kaputt

8. (Wahrscheinlich noch viele mehr)

Wie geht es jetzt weiter?

Die diet libc und libowfat sind nur das Fundament. Wir brauchen mehr bloatfreie Bausteine. Die Welt braucht z.B. eine kleine Bourne Shell!

Die Welt hat genügend wir-linken-einfach-alles-rein Kleister-Pakete wie perl, python, tcl, mozilla, java. Gebraucht werden kleine Tools, die nur einen kleinen und wohldefinierten Job tun, den aber richtig gut.

Ich habe angefangen, einen LDAP-Server zu schreiben. Teil davon sind ASN.1-Routinen für DER-Encoding, die auch in einem echt kleinen SSL-Stack benutzt werden könnten.

Wenn du helfen willst

Die Homepage des Projektes ist http://www.fefe.de/dietlibc/.

Zuerst solltest du auf die Mailingliste kommen und die TODO-Datei aus dem CVS lesen.

Wenn du auch keine Lust hast, eine Testsuite zu schreiben, dann nützt du wahrscheinlich am meisten, wenn du bei embutils oder mininet hilfst. Ansonsten bleibt bei der diet libc eigentlich nicht mehr so viel zu tun neben Fehlersuche.

Abgesehen davon: bringe deinen Arbeitgeber dazu, eine Kommerz-Lizenz zu kaufen!

Oder hilf mir bei dem LDAP-Server!

Endlich! Die Letzte Folie!

Danke, daß ihr meinen Sermon angehört habt!

Für langsame Schreiber: `http://www.fefe.de/dietlibc/`

Noch Fragen?