GNU CLib

Nathanael Hübbe

nathanael.huebbe@informatik.uni-hamburg.de

Deutsches Klimarechenzentrum (DKRZ)

08-06-2011

1. Warum eine CLib?
   - Ein Beispiel, endlich erlärt
2. Einige Aufgaben der CLib
   - Speicherverwaltung
   - Dateizugriff
   - Prozesse & Environment
   - Häufig gebrauchte Funktionen
3. Geschichte der GNU CLib
4. Quellen

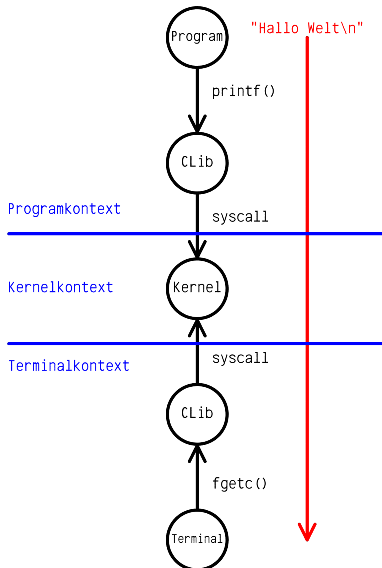
- Beispiel:

```
void main() {
    printf("Hallo␣Welt\n");
}
```

- Wirkung: Ausgabe von `"Hallo␣Welt\n"`
  - Aber: Angezeigt vom Terminal, nicht vom Program!
- Wie kann printf() das Terminal anweisen, `"Hallo␣Welt\n"` auszugeben?

- Was wirklich passiert:
  - printf() ist eine Funktion der CLib.
    - Ist immer vorhanden,
      denn die CLib gehört zum Sprachstandart.
  - printf() macht einen Syscall.
    - Kontextwechsel!
    - Der Kernel wird angewiesen, `"Hallo␣Welt\n"` in die Datei mit dem Dateidescriptor 1 zu schreiben.
    - Die CLib ist von der Kernelversion abhängig!
  - Der Kernel stellt fest, dass der Descriptor 1 eine Pipe bezeichnet. Am anderen Ende wartet das Terminal.
    - Dafür hat das Terminal gesorgt, als es den Kernel angewiesen hat, den Hallo-Welt-Prozess zu starten.
  - Der Kernel weckt das Terminal auf
    und überreicht `"Hallo␣Welt\n"` als Eingabe.
  - Das Terminal weist den XServer an,
    `"Hallo␣Welt\n"` darzustellen...

Ein Beispiel, endlich erlärt



Die CLib ist...

- das Bindeglied zwischen Programm & Kernel.
  - Wichtige Abstraktionsebene zur Entkopplung.
- die Bibliothek für alle grundlegenden Funktionen.

1 Warum eine CLib?
- Ein Beispiel, endlich erlärt

2 Einige Aufgaben der CLib
- Speicherverwaltung
- Dateizugriff
- Prozesse & Environment
- Häufig gebrauchte Funktionen

3 Geschichte der GNU CLib

4 Quellen

- void* malloc(size_t); und void free(void*);
  - Nicht immer ist ein Syscall notwendig.
- void* realloc(void*, size_t);
  - Kann den Block umkopieren
    $\Rightarrow$ Pointer auf den Block werden ungültig!
- void* calloc(size_t count, size_t elementSize);
  - Füllt den Block mit Nullen.
- Viele weitere Features, z. B. zum Debuggen, Obstacks, ...

- I/O-Streams:
  - Alle Funktionen beginnen mit ’f’.
  - 3 Standartstreams: stdin, stdout & stderr.
    - Deklariert als FILE*
  - Highlevel-Interface das auf den Funktionen des Lowlevel-Interface aufbaut.
  - FILE* fopen(const char* filename, const char* mode);
    int fclose(FILE*);
    - Auch in 64-Bit Variante verfügbar.

- Auf Streams schreiben:
  - ```
    int fputc(int c, FILE* stream);
    int putchar(int c);
    ```
  - ```
    int fputs(const char* s, FILE* stream);
    int puts(const char* s);
    ```
  - ```
    size_t fwrite(const void* data, size_t size,
                  size_t count, FILE* stream);
    ```
  - ```
    int fprintf(FILE* stream, const char* template, ...);
    int printf(const char* template, ...);
    ```
  - ```
    int asprintf(char** ptr, const char* template, ...);
    ```
    ```

- Von Streams lesen:
  - Analog zum Schreiben.
  - + Unreading.
    - Inputstreams sind Stacks!
      Fassen aber nur 1 Byte in der GCLib
    - `int ungetc(int c, FILE* stream);`
    - Beispiel:

```
char a = (char)fgetc(stdin), b;
ungetc(a+1, stdin);
b = (char)fgetc(stdin);
assert(b == a+1);
```

- Lowlevel Interface:
  - Dateideskriptoren sind ints.
  - int open(const char* path, int flags, mode_t mode);
    int close(int descriptor);
  - ssize_t read(int descriptor, void* buffer, size_t size);
    ssize_t write(int descriptor, const void* buffer, size_t size);
  - off_t lseek(int descriptor, off_t offset, int whence);
  - FILE* fdopen(int descriptor, const char* mode);

- int system(const char* command);
- FILE* popen(const char* command, const char* mode)
  - int pclose(FILE*) wartet auf Beendigung.
- Weitere Möglichkeit: fork() und die exec() Familie.
  - Erlaubt auch explizit Environment-Variablen zu setzen.

- String- & Array-Handling
  - memcpy(), memmove(), memset(),
    die entsprechenden String-Varianten und strncat()
  - memcmp(), memchr(), memmem()
    und viele weitere Varianten zum Suchen.
  - char∗ strfry(char∗);
    ⇒ Es gibt eine Funktion für praktisch jeden Zweck,
    ein Blick in die Dokumentation lohnt immer!
- Mathematik
  - Makros für Konstanten, z. B. M_PI, M_PI_2, M_2_SQRTPI, ...
  - Trigonometrische Funktionen
    inklusive inverser Funktionen.
  - Exponentialfunktionen & Logarithmen.
  - Hyperbolische Funktionen (z. B. Katenoide: cosh()).

1 Warum eine CLib?
- Ein Beispiel, endlich erlärt

2 Einige Aufgaben der CLib
- Speicherverwaltung
- Dateizugriff
- Prozesse & Environment
- Häufig gebrauchte Funktionen

3 Geschichte der GNU CLib

4 Quellen

- Mitte der **1980er**
  Begin der Arbeit von Roland McGrath.
- **1985**
  Gründung der Free Software Foundation durch Richard Stallman.
- **1988**
  ANSI C fast vollständig implementiert.
- **1992**
  ANSI C-1989 & POSIX.1-1990
- Frühe **1990er**
  Linux-Zweig mit Versionen 2 bis 5.
- **1997**
  glibc 2.0: Besserer POSIX Support als Linux-Zweig.
  $\Rightarrow$ Aufgabe des Linux-Zweigs,
  libc.so.6 == glibc 2 unter Linux.

1 Warum eine CLib?
- Ein Beispiel, endlich erlärt

2 Einige Aufgaben der CLib
- Speicherverwaltung
- Dateizugriff
- Prozesse & Environment
- Häufig gebrauchte Funktionen

3 Geschichte der GNU CLib

4 Quellen

- **GNU C Library Website**
  http://www.gnu.org/software/libc/
- **GNU C Library Manual**
  http://www.gnu.org/software/libc/manual/
- J. Quade, E. Kunst: **Linux-Treiber entwickeln**
  3. Auflage dpunkt.verlag, ISBN 978-3-89864-696-3
- **Wikipedia**
  http://en.wikipedia.org/

Fragen?