Einführung Systemprogrammierung

(Vorläufige interne Fassung, wird überarbeitet)

4.4.00, 18.4.00
Uni Weimar, G. Schatter

Was ist Systemprogrammierung?

	Anwendungs-Programmierung	*System-Programmierung*
Zweck	Lösung von numerischen und nichtnumerischen Verarbeitungsaufgaben: Resultat	Lösung von Kommunikationsaufgaben mit Hardware bzw. mit Prozessen: Operation
Mittel	Kern der Programmiersprachen bzw. Bibliotheken	Bibliotheken der Programmiersprachen
Ergebnis	numerisches oder nichtnumerisches Resultat i. a. einer Verknüpfungsoperation von Eingabeparametern	Vollzug einer Handlung, Resultat i.a. nur zur Vollzugskontrolle oder Fehlerauswertung wichtig
Forderung	effizient, Sortiment von Datentypen, mächtige Befehle	hardwarenah, schnell, flexibel, einfach

im weiteren Sinne:
Nutzung aller Schnittstellen des Betriebssystems (BS, OS) durch Programmierung zur Resourcenverwaltung

im engeren Sinne:
Nutzung der BS-Primitive auf niedrigstem BS-Niveau

Warum SP?

schnelle Einarbeitung in spez. Systeme: MVS, VM, BS2000, UNIX: Solaris, HP-UX, Linux..., Windows 98/NT, MacOS
Strukturierung komplexer Programmsysteme (Unterteilung in interag. Komponenten)
Konzeption und Implementierung spezialisierter Systeme (eingebette Systeme, Automatisierung)
Fehlertolerante Systeme
Verständnis für Ablauf in BS
- Ökonom. Nutzung Hardware
- Laufzeitoptimierung
- Verwaltung, Fehlersuche

Welche Schnittstellen haben BS für die Programmierung?

a)
Shell: Kommandozeileninterpreter, dh. Kommunikation mit BS über Kommandos, unter Unix nur mässig standardisiert, man (1)
Normalfall: Tastatur
File: Shell-Skript über Editor entwickeln

b)
Programmierschnittstelle zum System = Funktionsrufe man (2), (3)
Eintrittspunkte:

system call, Systemaufrufe	*Funktionen Standard Bibliothek C, Bibliotheksfunktionen*
Dienste für Programme: File öffnen, lesen ProgrammstartSpeicher reservieren Zeit lesen	Bibliothek liefert Funktionen: Variable formatieren für I/O Stringvergleich
man pages (2)	man pages (3)
service points zur Dienstanforderung vom Kern ca. 50-100 calls, kann nicht beeinflusst werden Definition in C statt Assembler	beeinflussen Kern nicht
minimale Funktion und Schnittstelle	ausführliche Funktion und Schnittstelle
Systemkern-Schnittstellen	keine Schnittstellen zumSystemkern, auch wenn sie selbst Systemfunktionen aufrufen

Unterschied call/function ist für BS-Designer fundamental, für Nutzer ist Unterschied nebensächlich
beide erscheinen als normale C-Funktionen, Funktionen sind ersetzbar - calls nicht!

(Skizze einfügen)

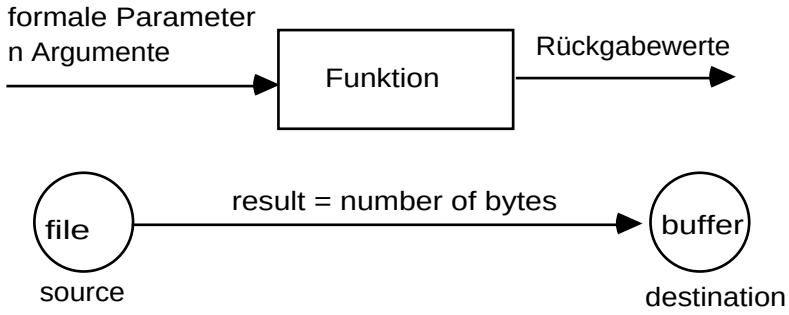
Nomenklatur Funktionen

result = function (parameter-liste);

result = read (source, destination, number);

result:

> 0 number of bytes (ok)
-1 error, Fehlerbehandlung



Funktionsprototypen allgemein:

```
#include <header.h>
typ Rückgabewert Funktionsname(Typ Argument1, ... , Typ Argument n);
                                                    Returns: ... if OK, ... on error
```

Funktionsprototypen speziell z. B.:

```
#include <unistd.h>
ssize_t read(int filedes, void *buff, size_t nbytes);
                                    Returns: #bytes read if OK, 0 if end of file, -1 on error
```

Einsetzbare Sprachen

Sprache	*Suffix*	*rel. Laufzeit*
Assembler	.s	1
C	.c	1,4 ... 2,5
Pascal	.p	2 ... 5
Fortran	.f	3,5 ... 7
Cobol		6,5 ... 12
Basic		10 ...20

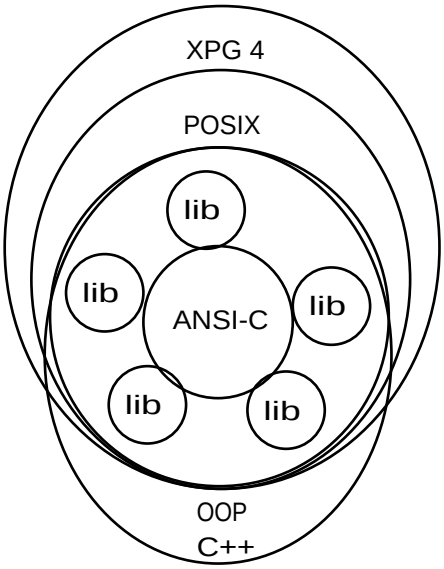
.h präprozessorfile, include-files
.o objektfiles
ausführbare Programme ohne Suffix!

Alternativen für die Systemprogrammierung:
PL/1
Modula-2
Ada
Assembler

Zwischenstellung C

zwischen höherer Programmiersprache und Assemblernähe
geglückter Versuch Vorteile beider Konzepte zu vereinen

Nähe zu Assembler	*Nähe zu höherer Programmiersprache HLL*
Registerzugriff	Datentypen hardwareunabhängig
Bitverarbeitung	algorithmische Steuerungsmöglichkeiten (Do, IF, WHILE...)
betriebssystemnahe Schnittstellen (UNIX-orientiert)	Blockkonzepte, scope-Bereiche
POSIX-Nähe	komplexe Ausdrücke



(Skizze einfügen)

Bedeutung C:
UNIX!
gut für arithmetische Probleme
E/A umfangreich
viel Software vorhanden
auf allen Rechnern verfügbar, UNIX i. a. standardmässig (ausser Sun)
Makroprozessor gut
Übergang zu OOP C++ möglich

aber auch Nachteile...
setzt algorithmisches Denken voraus, das an einer "sauberen" Sprache gelernt sein sollte
Orthogonalität mangelhaft
Hardwareabhängigkeiten vorhanden
Normierung nicht streng
Operatoren-Assoziativität nicht einheitlich

Geschichte und Einordnung, Standardisierung

C eng mit Unix verbunden
Anfang 70er Jahre Ken Thompson, Dennis Ritchie Bell Labs

BCPL	60er Richard	typenlose Sprache Nähe Assembler
B	70, Thompson	typenlose Sprache Nähe Assembler
C	72, Dennies, Ritchie	vollständige Sprache für strukturiertes Programmieren, Datentypen
ANSI-C	84	ANSI_84
X/OPEN C	85	UNIX V
C++	85 Stroustrup	kommerzieller Compiler von AT&T
XPG4	92	Xopen Portability Guide

C ist echte Untermenge von C++

Unix > 95% in C geschrieben
UNIX V6 (1973) K&R-C (Kernighan/Ritchie)

Entwicklung zu BS-unabhängiger Sprache
Dialekte 12.1988 ANSI Sprachbeschreibung und Standardisiereung

Eigenschaften C

Angebote der Sprache C
- Typenkonzept
- Blockstruktur
- hardwarenahe Programmierung maschinenunabhängig
- Makro-Präprozessor
- Standard-Include-Dateien
- Rekursive Funktionen
- separate Compilierung

- Systemaufrufe
- Operationen Register- und Bitebene
- reichhaltige Library
- Typkonvertierungen relativ ungehindert, daher relativ "unsicher"

Nicht enthalten, aber in Bibliotheksfunktionen bzw. Betriebssystemfunktionen
- Operationen auf Datentyp array
- Ein-/Ausgabe-Routinen
- mathematische Funktionen
- Prozeßkooperation (Kommunikation, Synchronisation)
- Heapverwaltung mit Garbage-Collektion

kleiner Sprachumfang -> Compiler kompakt und schnell
Strukturierte Programmierung:
C formatfreie Sprache, keine festen Spaltenpositionen, freies Einrücken
leicht lesbar, wartrungsfreundlich
Kontrollstrukturen reichhaltig, Sprungbefehle nicht nötig

modulare Programmierung:
logisch zusammengehörige Informationen in eigenen Quellcode-moduln (Dateien) zusammenfassen -> zu bindefähigen Objectcodemoduln machen (Typdefinitionen, Datendefinitionen, Funktionsdefinitionen, Makrodefinitionen..)
portable Programmierung
defensive Programmierung
kompakte Programmierung
mächtige Sprache, z. B. beste direkte Speichermanipulation

Einsatz C

Systemprogrammierung
systemnahe Programmierung
Assembler, Compiler, Betriebssysteme, Programmierumgebung, Testsysteme ...
technisch-wissenschaftliche Software: Grafik, Prozeßsteuerung, Simulation
"beste prozedurale Hochsprache für professionalelle Softwareentwicklung"

Werkzeuge

Dokumentation (man-pages)
Shells
Texteditoren
C-Compiler und Utilities
lint
make
Debugger
Compilerbau Werkzeuge lex, yacc

Weiterführung

Objective C, C++, Java (Vereinfachungen von C++)

Ablauf

der Weg vom Programmtext - Binärfile:

Schritt	*Werkzeug*	*Resultat*
Editieren	Texteditor,	evtl. mehrere Quelldateien
Übersetzen	Compiler, dabei zuerst Präprozessor	Objektdatei, Modul (Maschinencode und Binder-Infos (linker))
Binden	Linker	lauffähiges Programm
ausführen		

(Skizze einfügen)

Elemente

Standardfunktionen: fertige und übersetzte Funktionen, die zu C gehören, linker bindet sie zu anderen Funktionen
jede Funktion muss deklariert werden, auch Standardfunktionen
Deklaration von übersetzten Funktionen in header-Dateien (Deklarationsdatei)
E/A z.B. in stdio.h
#include <stdio.h>
...

Funktionen sind Bausteine aus denen C-Programm zusamengesetzt
führen Aufgaben aus
i.a. Eingabe- und Ausgabewerte (Rückgabe)
selbst Hauptprogramm eine Funktion (Vorgeschrieben : main)
Rückgabewert main-Funktion auch exit-status

Argumentübergabe

(Skizze einfügen)

argc
argument count, int-Parameter, der bei Programmaufruf Anzahl übergebeber Zeichenfolgen enthält, incl. Programmnamen selbst

argv
argument vector, ist Vektor von Zeigern auf die im programmaufruf angegebeben Zeichenfolgen
argv[0] zeigt auf Programmnamen selbst, argv[1] auf ersten Parameter

Rückgabewerte
bis auf Funktionen bei denen Rückgabewert explizit Typ void vereinbart wurde, geben alle Funktionen Wert an aufrufer zurück
typ Rückgabewert wird im Funktionsprototyp und Funktionsdefinition festgelegt (Standard ist int)
Rückgabewert wird in return-Zeile übermittelt

Prozeduren sind Funktione die keine Wert zurückliefern (und mit void vereinbart wurden)

Ein-Ausgabe

nicht in C definiert
alle E/A mit Funktionen
Standardfunktionen in stdio.h definiert
Begriff Standardeingabe und -ausgabe (Display, keyboard)

High- level-Dateizugriff: gepufferte EA
low-level: ungepuffert

File-Zeiger standardmässig definiert
stdout
stdin
stderr

zugehörige Gerätedateien (streams) werden bei Programmstart automatisch geöffnet

Präprozessor

(Skizze einfügen)

Funktion Headerdatei:
Sammelplatz für Definition von
- Konstanten,
- Makro
- Datentyp
- Funktionen

für Dateien. Präprozessor ersetzt Zeile durch vollständigen Text.

vor Compiler:
- einfügen weitere Quelltextdateien und Bibliotheken, bedingte Übersetzung
- Makrodefinition und -ersetzung
- Compileroptionen setzen

Steuerung über Direktive
Aufbau: #

Dateien einfügen
#include (header-, include-Dateien)
Präprozessorfiles als:

allg. Header	< .h>	/usr/include bzw. /usr/local/include
hardwarenahe H..	<sys/ .h>	/usr/include/sys
nutzereigene H.	" .h"	aktuelles Verzeichnis bzw. usr/include

2 Schreibweisen, die Suchpfad bestimmen:

#define : als Makro definiert werden

Makros
Abkürzung von Zeichenfolgen oder Benennung von Konstanten
#define Makroname Ersatztext
symbolische Konstanten : Grossbuchstaben

Header und Bibliotheken

Header stellen bereit:
Makros, Datentypen, globale Variable, Finktionen, ANSI-C-Bibliotheken

Linker
In Kommandozeile Bibliothek angeben (bis auf Standardbibliothek)

Standardbibliotheken

Funktionen, die ANSI-Standardimplementierung gehören
sind aber keine Elemente von C!, Deklaration in ANSI-C festgelegt (Portabilität)
15 Header von ANSI-C:

`assert.h`	assert-Makro
`ctype.h`	Test, Umwandlung von characters
`errno.h`	Fehlerauswertung
`float.h`	Makros für Behandlung Fliesskommazahlen
`limits.h`	Definition Systemgrenzen, -schranken
`locale.h`	Funktion setlocale()
`math.h`	trigonom. , hyperbol., expon-log., Potenzfkt., Zahlenanteile, Absolut für double
`setjmp.h`	Funktionen setjmp(), longjmp()
`signal.h`	Signalbehandlung
`stdarg.h`	Funktionen und Makros für variable Argumentlisten
`stddef.h`	Definitionen ptrdiff_t, NULL, size_t, wchart_t, offsetpf, errno
`stdio.h`	Dateioperationen, Zugriffsfkt., formatierte EA, Character-EA, direkte EA, Dateipositionierung, Fehlerbehandlung
`stdlib.h`	Schnittstelle zum BS, Umwandlung, Zufallszahlen, Speicherverwaltung, suchen + sortieren, Integer-Arithmetik
`string.h`	Kopier, Vergleich, Suche: Zeichenfolgen (strings)
`time.h`	Zeitmanipulation time(), ctime(), strftime() ...

Achtung – POSIX Header-files!

`dirent.h`	
`fcntl.h`	
`grp.h`	
`pwd.h`	
`setjmp.h`	
`signal.h`	kill() und sig...()
`stdio.h`	
`sys/stat.h`	
`sys/times.h`	BS-abhängige Typdefinitionen
`sys/utsname.h`	
`sys/wait.h`	
`termios.h`	
`time.h`	
`utime.h`	
`unistd.h`	alle POSIX-Funktionen die nicht oben definiert sind, z. B.: fork(), read() ..

Achtung – POSIX Datentypen aus <`sys/types.h`>

`dev_t`	Gerätenummer
`gid_t`	Gruppen-id
`ino_t`	Sewriennumer von files, Inodenummer
`mode_t`	Fileattribute
`nlink_t`	Hardlinkzähler
`off_t`	Filegrößen
`pid_t`	Prozeß-ID

`size_t`	entspricht ANSI-C size_t
`ssize_t`	Anzahl von bytes oder -1
`uid_t`	User-ID

Programmierstil

Regeln Softwareengineering beachten
- Konventionen achten (Sprache, Namen, Datentypen, include-files)
- Allgemeinheit, Verständlichkeit

allgemeine Forderung an Sofwarequalität beachten:
korrekt, robust, erweiterbar, wiederverwendbar, kommentiert

Defensive Programmierung:
alle potenziellen Fehler abfangen und bearbeiten
Test auf Nullzeiger vor Dereferenzierung
Vermeidung von statischen limits oder Verwendung von Systemlimits
Vermeidung von Speicherlöchern

Modularisierung:
Zerlegung in funktionale Module
Aufruf der Module über klare Schnittstellen
Trennung der Module in einzelne Code- und Headerdateien
überblickbare Teilfunktionen
klare Gliederung auch optisch
Code soll auch beimÜberfliegen verständlich sein
Formatierung soll Lesbarkeit unterstützen

Effizienz:
Wiederverwendung erhaltener Werte
sparsamer Aufruf von systemfunktionen
sparsamer Umgang mit systemressourcen
unnötige Kopieraktionen vermeiden

Kommentare und Dokumentation:
klar, kurz, eindeutig
ausführlich nur vor komplexen Codeteilen
keine Wiedrholung/Vorwegnahme Funktionsdeklarationen
Grundkonzept detailliert erklären
insbes, Programmier-, Modulschnittstellen dokumentieren
Code nicht verbal wiederholen
sprachlich angemessen, Idiome weitgehend vermeiden