

Betriebssysteme 2 | BSys2

Zusammenfassung

```
int *p = &x; // &x = Adresse des ints, * = Pointer-Bezeichner
int y = *p; // *p = Wert einer int-Adresse, y = 5, * = Dereferenzierer.
```

4096	2048	1024	512	256	128	64	32	16	8	4	2	1
2 ¹²	2 ¹¹	2 ¹⁰	2 ⁹	2 ⁸	2 ⁷	2 ⁶	2 ⁵	2 ⁴	2 ³	2 ²	2 ¹	2 ⁰
100,00	800,0	400,0	200,0	100,0	60,0	40,0	20,0	10,0	8,0	4,0	2,0	1,0
1'048'576	65'536	4'096	256	16	1							
16 ⁶	16 ⁴	16 ³	16 ²	16 ¹	16 ⁰							
10 00 00,	01 00 00,	00 10 00,	00 01 00,	00 00 10,	00 00 01,							

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

1. BETRIEBSSYSTEM API

Aufgaben: Abstraktion, Portabilität, Ressourcenmanagement & Isolation der Anwendungen, Benutzerverwaltung und Sicherheit.

Privilege Levels: *Kernel-Mode* (darf alles ausführen, Ring 0), *User-Mode* (darf nur beschränkte Menge an

Instruktionen ausführen, Ring-Mode)

Kernels: *Microkernel* (nur kritische Teile laufen im Kernel-Mode), *Monolithisch* (meiste OS, weniger Wechsel,

weniger Schichten, *Unikernel* (Kernel ist nur ein Programm)

syscall = Aufruf eines Systemaufrufs, im Kernel-Mode zu machen, jedes OS-Kernel-Funktion hat

einen Code, der dem Register übergeben werden muss. (x86 hat den Code 0x)

ABI: Application Binary Interface, Abstrakte Schnittstelle mit platformunabhängigen Aspekten.

API: Application Programming Interface, Konkrete Schnittstellen, Calling Convention, Abbildung

von Datenstrukturen. *Linux-Kernels* sind API-, aber nicht ABI-kompatibel. (C-Wrapper-Funktionen)

POSIX: Portable Operating System Interface. Sammlung von IEEE Standards, welche die Kompatibilität

zwischen OS gewährleistet. Windows ist nicht POSIX-konform.

1.1. PROGRAMMARGUMENTE

clang -o abc.c -o abc.c : Die Shell teilt Programmargumente in Strings auf (trennung durch Leerzeichen, sonst Quotes). **Calling Convention** : OS schreibt Argumente als null-terminierte Strings in den Speicherbereich des Programms. Zusätzlich legt das OS ein Array argv an, dessen Elemente jeweils auf das erste Zeichen eines Arguments zeigen. Die Art und Weise, wie das gehandhabt wird, ist die Calling Convention. Werden explizit angegeben, nützlich für Informationen, die bei jedem Aufruf anders sind.

```
int main(int argc, char **argv) { ... } // argv[0] = program path
```

1.2. UMGEBUNGSVARIABLEN

Strings, die mindestens ein Key-Value enthalten **OTHER=1, PATH=/home/js/bin**. Der Key muss eindeutig sein. Unter POSIX verwendet das OS die Umgebungsvariablen innerhalb jedes laufenden Prozesses. Werden initial festgelegt. Das OS legt die Variablen als ein null-terminiertes Array von Pointern auf null-terminierte Strings ab. Unter C zeigt die Variable **extern char **environ** darauf. Sollte nur über unterstehende Funktion manipuliert werden. Werden implizit bereitgestellt, nützlich für Informationen, die bei jedem Aufruf gleich sind.

- **Abrufen einer Umgebungsvariable:** `char *value = getenv("PATH");`
- **Setzen einer Umgebungsvariable:** `int ret = setenv("HOME", "/usr/home/1");`
- **Entfernen einer Umgebungsvariable:** `int ret = unsetenv("HOME");`
- **Hinzufügen einer Umgebungsvariable:** `int ret = putenv("HOME:/usr/home/");` (gefährlich wegen Pointer auf NULL)

Größere Konfigurationsinformationen sollten über **Dateien** übermittelt werden.

2. DATEISYSTEM API

Anwendungen dürfen nie annehmen, dass Daten gültig sind.

Arbeitsverzeichnis: Bezugspunkt für relative Pfade, jeder Prozess hat eines

(getcwd(), chdir(): nimmt String, fehler:1 (nicht File Deskriptor).

PADE: Absolut (beginnt mit /), Relativ (beginnt nicht mit /), Kanonisch (Absolut, ohne . und ..), realpath()

– **NAME_MAX:** Maximale Länge eines Dateinamens (exklusive terminierender null)

– **PATH_MAX:** Maximale Länge eines Pfades (inklusive terminierender null) beinhaltet: Wert von NAME_MAX,

– **POSIX_NAME_MAX:** Minimaler Wert von NAME_MAX nach POSIX (40)

– **POSIX_PATH_MAX:** Minimaler Wert von PATH_MAX nach POSIX (256)

// Gibt Arbeitsverzeichnis aus

```
int main (int argc, char **argv) { char *wd = malloc(PATH_MAX);
```

```
getcwd(wd, PATH_MAX); printf("Current WD is %s", wd); free(wd); return 0; }
```

Zugriffrechte: Je 3 Permission-Bits für Owner, Gruppe und andere Benutzer. Bits sind: read, write, execute; n=4, w=2, x=1. **Beispiel:** 0740 oder rwx r-- -- (Owner hat alle Rechte, Group kann lesen, andere haben keine Rechte). **POSIX:** `S_IRWXU = 0700, S_IRUSR = 0200, S_IRGRP = 0040, S_IXOTH = 0001. Werden mit | verknüpft.`

POSIX-API: für direkten Zugriff, alle Dateien sind ro-Binärdateien. **C-API:** für direkten Zugriff auf Streams. **POSIX FILE API:** für direkten, unformatierten Zugriff auf Inhalt der Datei. Nur für Binärdateien verwenden. **errno:** Makro oder globale Variable vom typ int. Sollte direkt nach Auftreten eines Fehlers aufgerufen werden.

```
if (chdir("docs") < 0) { if (errno == EACCESS) { printf("Error: Denied"); }}
```

strenv() gibt die Adresse eines Strings zurück, der den Fehlercode code textuell beschreibt. Person schreibt text gefolgt von einem Doppelpunkt und vom Ergebnis von strenv() (errno) auf den Errorstring.

2.1. FILE-DESCRIPTOR (FD)

Files werden in der POSIX-API über FD's repräsentiert. Gibt nur innerhalb eines Prozesses. Return **Index in** **Index** auf geöffnete Dateien im Prozess → Enthält **Index in systemweite Tabelle** → Enthält Daten zur Identifikation der Datei. **STDIN_FILENO** = 0: standard input, **STDOUT_FILENO** = 1: standard output, **STDERR_FILENO** = 2: standard error

```
int open (char *path, int flags, ...); // öffnet eine Datei. Erzeugt FD auf Datei an path.
```

Flags gibt an, wie die Datei geöffnet werden soll:

– **O_RDONLY:** nur Lesen

– **O_RDWR:** Lesen und schreiben

– **O_CREAT:** Erzeuge Datei, wenn sie nicht existiert,

– **O_APPEND:** Setze Offset als Ende der Datei vor jedem Schreibzugriff

– **O_TRUNC:** Setze Länge der Datei auf 0

int close (int fd): schließt Datei bzw. dealloziert den FD. Kann dann wieder für andere Dateien verwendet werden. Wenn FD's nicht geschlossen werden, kann das File-Limit erreicht werden, dann können keine weiteren Dateien mehr geöffnet werden. Wenn mehrere FD's die gleiche Datei öffnen, können sie sich gegenseitig Daten überschreiben.

```
int fd = open("myfile.dat", O_RDONLY);
```

```
if (fd < 0) { /* error handling */ } /* read data: */ close(fd);
```

sizeof_t read(int fd, void *buffer, size_t n):

kopiert die nächsten n Bytes am aktuellen Offset von **fd** in den **Buffer**.

sizeof_t write(int fd, void *buffer, size_t n):

kopiert die nächsten n Bytes vom **buffer** an den aktuellen Offset von **fd**

```
#define N 32
char buf[N]
char *path(PATH_MAX); // source path
char *dpath(PATH_MAX); // destination path
// ... gets paths from somewhere
int arc = open(spath, O_RDONLY);
int read = open(dpath, O_WRONLY | O_CREAT, S_IRWXU);
sizeof_t read_bytes = read(arc, buf, N);
write(dbuf, read_bytes); // if file closed early, use return value
close(arc); // of "read_bytes"
close(dbuf);
```

off_t lseek(int fd, off_t offset, int origin): *Springen in einer Datei.* Verschiebt den Offset und gibt den neuen Offset zurück. **SEEK_SET:** Beginn der Datei, **SEEK_CUR:** aktueller Offset, **SEEK_END:** Ende der Datei. **lseek(fd, 0, SEEK_CUR)** gibt aktuellen Offset zurück, **lseek(fd, 0, SEEK_END)** gibt die Größe der Datei zurück.

sizeof_t read/write(int fd, void *buffer, size_t n, off_t offset): *Lesen und schreiben ohne Offsetänderung.* Wie read bzw. write. Statt des Offsets von fd wird der zusätzliche Parameter offset verwendet.

2.1.1. Unterschiedliche Windows und POSIX

Bestandteile von Pfaden werden durch **Backslash (\)** getrennt, ein **Wurzelverzeichnis** pro Datenring/Partition, andere File-Handling-Funktionen.

2.2. C STREAM API

Unabhängig vom Betriebssystem, Stream-basiert, gepuffert oder ungepuffert, hat einen eigenen File-Position-Indikator.

Streams: FILE + enthält *Informationen über einen Stream*. Soll nicht direkt verwendet oder kopiert werden, sondern nur über C-API erzeugte Pointer.

FILE + fopen(char const *path, char const *mode): *Öffnen eine Datei.* Erzeugt FILE-Objekt für Datei an path. Flags für mode: "r" (Datei lesen), "w" (in neue oder bestehende Datei schreiben),

"a" (in neue oder bestehende Datei anfügen), "r+" (Datei lesen & schreiben), "w+" (neue oder gelöschte bestehende Datei lesen & überschreiben), "a+" (neue oder bestehende Datei lesen & an Datei anfügen). Gibt Pointer auf

erzeugtes FILE-Objekt zurück oder 0 bei Fehler. **FILE + fopen(int fd, char const * mode)** ist gleich, aber statt Pfad wird direkt der FD übergeben. **int fclose (FILE *stream)** gibt FD zurück.

Nach API-Umwandlung vorherig nicht mehr verwenden.

int fclose(FILE *stream): Schließt eine Datei. Ruft **fclose()** (schreibt Inhalt aus Speicher in die Datei) auf,

schließt den Stream, entfernt fclose aus Speicher und gibt 0 zurück wenn OK, andernfalls EOF.

int fgetc(FILE *stream): Liest das nächste Byte und erhöht FI um 1.

char * fgets(char *buf, int n, FILE *stream) liest bis zu n - 1 Zeichen aus stream.

int ungetc(int c, FILE *stream): Lesen rückgängig machen. Nutzt Unget-Stack.

int fputs(int c, FILE *stream): Schreibe c in eine Datei. Int fputs(char *s, FILE *stream)

schreibt die Zeichen vom String s bis zum terminierenden 0 in stream.

2.2.1. Dateiteile und Fehler:

int ferror(FILE *stream) gibt 0 zurück, wenn Dateiteile **nicht** erreicht wurde

int ferror(FILE *stream) gibt 0 zurück, wenn kein Fehler auftrat.

2.2.2. Manipulation des File-Position-Indikator (FPI):

long ftell(FILE *stream) gibt den gegenwärtigen FI zurück, **int fseek (FILE *stream, long**

offset, int origin) setzt den FPI, analog zu lseek, **int rewind (FILE *stream)** setzt den Stream zurück.

3. PROZESSE

Prozesse (aktiv) sind die **Verwaltungseinheit** des OS für Programme (passiv). Jedem Prozess ist ein virtueller Adressraum zugeordnet.

Ein Prozess umfasst das **Abbild eines Programms** im Hauptspeicher (text section), die **globalen**

Variablen des Programms (data section), Speicher für den Heap und Speicher für den Stack.

Process Control Block (PCB): Das Betriebssystem hält Daten über jeden Prozess in jeweils einem

PCB vor. Speicher für alle Daten, die das OS benötigt, um die Ausführung des Prozesses ins

Gesamtssystem zu integrieren, u.a.: Diverse IDs, Speicher für Zustand, Scheduling-Informationen, Daten zur Synchronisation, Security-Informationen etc.

Interrupts: Kontext des aktuellen Prozesses muss im dazugehörigen PCB gespeichert werden (context save): Register, Register, Interrupt Pointer, MMU-Konfiguration. **Interrupt-Handler** überschreibt den Kontext. Anschließend wird Kontext aus PCB wiederhergestellt (context restore).

Prozess-Erstellung: Das OS erzeugt den Prozess und lädt das Programm in den Prozess. Unter POSIX getrennt, unter Windows eine einzige Funktion.

Unter POSIX hergestellt: Baumstruktur, startet bei Prozess 1.

3.1. PROZESS-API

pid_t fork(void) erzeugt **exakte Kopie** (C) als Kind des Prozesses, mit eigener Prozess-ID (> 0). Die Funktion führt in **beiden** Prozessen den Code an derselben Stelle fort.

void exit(int code): Beendet das Programm und gibt code zurück.

void wait(int *status): unterbricht Prozess, bis Child beendet wurde.

pid_t waitpid(pid_t pid, int *status, int options): wie wait(), aber pid bestimmt, auf welchen Child-Prozess man warten will (p = 0 = Prozess mit dieser ID, -1 = irgendeinen, 0 = alle C mit der gleichen Prozessgruppe ID).

```
void spawn_worker (...) {
```

```
if (fork() == 0) { /* do something in worker process; */ exit(0); }
```

```
} for (int i = 0; i < n; i++) { spawn_worker(...); }
```

```
// ... do something in parent process
```

```
do { pid = wait(0); } while (pid > 0) // errno == ECHILD; // wait for all children
```

exec()-Funktionen: Jede davon **ersetzt** im gerade laufenden Prozess das Programmimage **durch ein anderes**. Programmdateien müssen spezifiziert werden. (z.B. `./a.out`, `./a.out` - `./a.out`)

	Programmargumente als Liste	Programmargumente als Array
Angabe des Pfads	execve()	execve(C)
mit neuem Environment	execv()	execv(C)
mit altem Environment	execvp()	execvp(C)
Suche über PATH	execvp()	execvp(C)

3.1.1. Zombie- & Orphan-Prozesse

C ist zwischen seinem Ende und dem Aufruf von **wait()** durch P ein Zombie-Prozess. **Dauerhafter**

Zombie-Prozess: P ruft wegen Fehler **wait()** nie auf. **Orphan-Prozess:** P wird vor C beendet.

P kann somit nicht mehr auf C warten, was bei Beendigung von C in einem dauerhaften Zombie

resultiert. Wenn P beendet wird, werden deshalb alle C an Prozess mit pid=1 übertragen, der

wait() in einer Endlosschleife aufruft.

unsigned int sleep (unsigned int seconds): unterbricht Ausführung, bis eine Anzahl

Sekunden ungefähr vergangen ist. Gibt vom Schlaf noch vorhandene Sekunden zurück.

int atexit (void (*function)(void)): Registriert Funktionen für Aufräumarbeiten vor Ende.

pid_t getpid() / getppid() geben die (Parent-)Prozess-ID zurück.

4. PROGRAMME UND BIBLIOTHEKEN

C-Quelle → Präprozessor → Bereinigte C-Quelle → Compiler → Assembler-Datei →

Assembler → Objekt-Datei → Linker → Executable

Präprozessor: Die Ausgabe des Präprozessors ist eine reine C-Datei (Translation-unit) ohne Makros,

Kommentare oder Präprozessor-Direktiven. **Linker:** Der Linker verknüpft Objektdateien (und

statische Bibliotheken) zu Executables oder dynamischen Bibliotheken. **Loader:** lädt Executables

und eventuelle dynamische Bibliotheken dieser in den Hauptspeicher.

4.1. ELF (EXECUTABLE AND LINKING FORMAT)

Binär-Format, das Kompilate spezifiziert. Besteht aus **Linking View** (wichtig für Linker, für Object-Files und

Shared Objects) und **Execution View** (wichtig für Loader, für Programme und Shared Objects).

Struktur: Besteht aus **Header**, **Programm Header Table** (execution view), **Segmente** (execution view),

Section Header Table (linking view), **Sektionen** (linking view)

4.2. SEGMENTE UND SEKTIONEN

Segmente und **Sektionen** sind eine andere Einteilung für die gleichen Speicherbereiche. View des **Loaders** sind die **Segmente**, view des **Compilers** die **Sektionen**. Definieren "gleichartige" Daten. Der **Linker** verbindet zwischen beiden Views.

Header: Beschreibt den **Aufbau** der Datei. Typ: 32/64-bit, Encoding, Maschinenarchitektur, Entry-point,

Infos zu den Segmenten in PH und SHT.

Segmente/Program Header Table und Sektionen: Tabelle mit n Einträgen, jeder Eintrag (je 32

Bytes) beschreibt: ein Segment (Typ und Flags), Offset und Größe, virtuelle Adresse und Größe im Speicher - kann unterschiedlich zur Datengröße sein. Ist Verbindung zwischen Segmenten im RAM und im File. Definiert, wo

ein Segment liegt und wohin der Loader es im RAM laden soll.

Segmente werden vom **Loader** dynamisch zur **Laufzeit** verwendet.

Section Header Table und Sektionen: Tabelle mit n Einträgen (≠ n). Jeder Eintrag (je 40 Bytes)

beschreibt eine Sektion (Name, Section-Typ, Flags, Offset und Größe, ...). Werden vom **Linker** verwendet:

Verschnitt Sektionen und erzeugt aufeinander beim Laden gebunden

String-Table: Bereich in der Datei, der nacheinander **null-terminierte Strings** enthält. Strings werden

relativ zum Beginn der Tabelle referenziert.

Symbole & Symboltabelle: Die Symboltabelle enthält jeweils einen **Eintrag je Symbol** (16 Byte: 48

Byte, 48 Wert, 48 Größe, 48 Info).

4.3. BIBLIOTHEKEN

Statische Bibliotheken: Archive von Objekt-Dateien. Name: Lib<name>.a, referenziert wird nur

einmal. **Linker** behandelt statische Bibliothiken wie **mehrere Objekt-Dateien**. Ursprünglich gab es

nur statische Bibliotheken (Einfach zu implementieren, aber funktionalität fu).

Dynamische Bibliotheken: Linken erst zur Laufzeit bzw. Laufzeit des Programms. Höherer Aufwand,

jedoch austauschbar. Exakte enthält nur Referenz auf Bibliothek. **Vorteile:** Entkopplert Lebenszyklus,

Schnellere Ladezeiten durch Lazy Loading, Flexibler Funktionsumfang.

4.4. POSIX SHARED OBJECTS API

void *dlopen (char * filename, int mode): öffnet eine dynamische Bibliothek und gibt ein

Handle darauf zurück, das man weiter der folgenden Werte:

– **RTLD_NOW:** Alle Symbole werden beim Laden gebunden

– **RTLD_LAZY:** Symbole werden bei Bedarf gebunden

– **RTLD_GLOBAL:** Symbole können beim Binden anderer Objektdateien verwendet werden

– **RTLD_LOCAL:** Symbole können nicht für andere Objektdateien verwendet

void *dlsym (void *handle, char * name): gibt die Adresse des Symbols name aus der mit

handle bezeichneten **Bibliothek** zurück. Mehr Typinformationen (Variable? Funktion?)

int type "Func_t": is a address of a function with a int param and int return type

typedef int (*Func_t)(int);

handle = dlopen("libmyLib.so", RTLD_NOW); // open library

Func_t f = dlsym(handle, "my_func_t"); // write my_function addr. into a Func_t

int *i = dlsym(handle, "my_int"); // get address of "my_int"

(f)(int);** // call "my_function" with "my_int" as parameter

int dclose (void * handle): schließt das durch handle bezeichnete, zuvor geöffnete Objekt.

char * dlerror(): gibt Fehlermeldung als null-terminierten String zurück.

Konventionen: Shared Objects können **automatisch** bei Bedarf geladen werden. Der Linker verwendet

den Linker-Namen, der Loader verwendet den SO-Namen.

– **Linker-Name:** Lib + Bibliothekname + .so (z.B. libmyLib.so)

– **SO-Name:** Linker-Name + . + Versionsnummer (z.B. libmyLib.so.2)

– **Real-Name:** SO-Name + . + Versionsnummer (z.B. libmyLib.so.2.2)

Shared Objects: Heute alle Executables benötigen **zwei Shared Objects:** Libc.so: Standard C

library, Lib-Unix.so: ELF Shared Object library (Lad-Shared Objects und rekursiv alle Dependencies).

Implementierung dynamischer Bibliotheken: müssen verschierbar sein, mehrere müssen in den

gleichen Prozess geladen werden. Die Aufgabe des Linkers wird in den Loader bzw. Dynamic Linker

verschoben (Load Time Relocability)

4.5. SHARED MEMORY

Dynamische Bibliotheken sollen **Code zwischen Programmen** teilen. Code soll **nicht mehrfach** im

Speicher abgelegt werden. Mit Shared Memory kann jedes Programm eine eigene virtuelle Page für

den Code definieren. Diese werden auf denselben Frame im RAM gemaßt. Benötigt **Position-**

Independent Code (Adressen nur relativ zum Instruction Pointer, Prozessor muss relative Instruktionen anordnen).

Relative Moves via **Relative Calls:** Mittels Hilfsfunktion wird Rückadressierung in Register abgeleitet,

somit

