

Betriebssysteme 2 | BSys2

Zusammenfassung

```
int *px = &x; // &x = Adresse des x's, * = Pointer-Bezeichner
int y = *px; // *px = 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
1000	800s	400s	200s	100s	40s	20s	10s	8s	4s	2s	1s	
1'048'576	65'536	4'096	256	16	1							
16 ⁶	16 ⁴	16 ³	16 ²	16 ¹	16 ⁰							
10 00 00s	01 00 00s	00 10 00s	00 01 00s	00 00 10s	00 00 01s							

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

1. BETRIEBSYSTEMS 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 anstrukturen ausführen, Ring 3)

Kernels: Mikrokernel (nur kritische Teile laufen im Kernel-Mode), **Monolithisch** (meiste OS, weniger Wechsel, weniger Schichten, *Unikernel* (Kernel ist nur ein Programm)

syscall: verlässt den Prozess, ist im Kernel-Mode zu machen. Jede OS-Kernel-Funktion hat einen Code, der einen Register übergeben werden muss. (*hex* hat den Code 60)

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-Wrappers-Funktionen*)

POSIX: Portable Operating System Interface. Sammlung von IEEE Standards, welche die Kompatibilität zwischen OS gewährleistet. Windows ist nicht POSIX-konform.

1.1. PROGRAMMARGUMENTUMENTE

clang -o abc.c -o abc.o : 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 an, dessen Elemente jeweils auf die erste Zeile 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/<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.

- Abfragen 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. DATEISYSTEMS API

Anwendungen dürfen nie annehmen, dass Daten gültig sind.

Arbeitsverzeichnis: Bezugspunkt für relative Pfade, jeder Prozess hat einen

(*getcwd()*, *chdir()*: nimmt String, *chdir()*: nimmt File Descriptor).

Pfade: Absolut (beginnt mit /), Relativ (beginnt nicht mit /), Kanonisch (Absolut, ohne „..“ und „.“, *realpath()*)

NAME_MAX: Maximale Länge eines Dateinamens (exklusiv 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 (*14*)

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

Zugriffrechte: je 3 Permission-Bits für Owner, Gruppe und anderen Benutzer. Bits sind: read, write, execute: *r=4, w=2, x=1* Beispiel: 0740 oder *rwx* *r=* *rw* *---* (Owner hat alle Rechte, Gruppe kann lesen, anderen haben keine Rechte). **POSIX:** *S_IRWXU = 0700, S_IRUSR = 0200, S_IRGRP = 0400, S_IXOTH = 0001*. Werden mit | verknüpft.

POSIX-API: für direkten Zugriff, alle Dateien sind ro-Binärdateien. **C-API:** für direkten Zugriff über 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 Errors werden.

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

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

2.1. FILE-DESCRIPTOR (FD)

Files werden in der POSIX-API über FD's repräsentiert. Gibt nur innerhalb eines Prozesses. Return Index in Tabelle aller geöffneten 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_WRONLY:* Lesen und Schreiben,
- O_CREAT:* Erzeuge Datei, wenn sie nicht existiert,
- O_APPEND:* Setze Offset ans 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 FD-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 Byte vom buffer an den aktuellen Offset von fd

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

off_t, t_seek(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. **t_seek(fd, 0, SEEK_CUR)** gibt aktuellen Offset zurück, **t_seek(fd, 0, SEEK_END):** gibt die Größe der Datei zurück.

sizeof, t_read/pwrite(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. Unterschiedene Windows und POSIX

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

2.2. C STREAM API

Unabhängig vom Betriebssystem, Stream-basiert, gepuffert oder unpuffert, hat einen eigenen File-Position-Indicator.

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

FILE + fopen(char *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 geteilte bestehende Datei lesen & überschreiben), "a+" (neue oder bestehende Datei lesen & an Datei anfügen). Gibt Pointer auf erzeugtes FILE-Objekt zurück oder 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 vorherige nicht mehr verwenden.

int fclose(FILE *stream): Schließt eine Datei. Ruft fflush() (schreibt Inhalt aus Speicher in die Datei) auf, schließt den Stream, entfernt File aus Speicher und gibt 0 zurück wenn OK, andernfalls EOF. **int fgetc (FILE *stream):** Liest das nächste Byte und erhöht Pfad 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 fputc (int c, FILE *stream):** Schreibe c in eine Datei. **int fputs (char *s, FILE *stream)** schreibt die Zeichen vom String s bis zur terminierenden 0 in stream.

2.1.2. Dateiname und Fehler:

int feof (FILE *stream) gibt 0 zurück, wenn Dateiname noch nicht erreicht wurde

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

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

long ftell (FILE *stream) gibt den gegenwärtigen FPI zurück, **int fseek (FILE *stream, long offset, int origin)** setzt den FPI, analog zu seek, **int rewind (FILE *stream)** setzt den Stream zurück.

3. PROZESSE

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

Der 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. Der Heap wird für globale Variablen genutzt und der Stack für Variablen die nur während dem Funktionsaufruf existieren.

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 Gesamtsystem 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, Flags, Instruction Pointer, MMU-Konfiguration. *Interrupt-Handler:* über schreibt 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.

Prozess-Hierarchie: Baumstruktur, startet bei Prozess 1.

3.1. PROZESS-API

pid_t, fork(void) erzeugt exakte Kopie (C) als Kind des Prozesses (P), 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.

pid_t, 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 mit der gleichen Prozessgruppen-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. Programmargumente müssen spezifiziert werden. (*... lab late, ... v als Array*)

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

3.1.1. Zombie- & Orphan-Prozesse
C zwischen seinem Ende und dem Aufruf von wait() durch P ein Zombie-Prozess. Dauerhaftes C zwischen seinem Ende und dem Aufruf von wait() auf. Orphan-Prozess: P wird vor C beendet. P kann somit nicht mehr auf C warten, was bei Beendigung von C in einem dauerhaftes Zombie resultiert. Wenn P beendet wird, werden deshalb alle C an Prozess mit pid übertragen, der wait() in einer Endlos-Schleife aufruft.

unsigned int sleep (unsigned int seconds): unterbricht Ausführung, bis eine Anzahl Sekunden ungefähr verstrichen ist. Gibt vom Schlaf noch vorhandene Sekunden zurück.

int atexit (void *function(void)): registriert Funktionen für Aufräumarbeiten vor Ende. **pid_t, getpid()** (**getpid()** 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 Objekt-Dateien (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)

Binary-Format, das Kompile 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. Laden Daten in den Speicher und definieren "gleichartige" Daten der Linker vermittelt zwischen beiden Views.

Header: Beschreibt den Aufbau der Datei. Typ: 32/64-bit, Encoding, Maschinenarchitektur, Entry-point, Infos zu den Einträgen in PHT und SHT.

Segment/Program Header Table und Sektionen: Tabelle mit n Einträgen, jeder Eintrag (je 32 Byte) beschreibt ein Segment Typ und Offset. Original-Adresse und Offset 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 m Einträgen (≠ n). Jeder Eintrag (je 40 Byte) beschreibt eine Sektion (Name, Section-Typ, Flags, Offset und Größe, ...). Werden vom Linker verwendet: Verschlüsselt Sektionen und erzeugt ausführbare Executable.

Strings: Alle Bytes in der Datei, der nacheinander null-terminierte Strings enthält. Strings werden relativ zum Beginn der Tabelle referenziert.

Symbol- & Symboltabelle: Die Symboltabelle enthält jeweils einen Eintrag je Symbol (16 Byte: 48 Name, 48 Wert, 48 Größe, 48 Info).

4.3. BIBLIOTHEKEN

Statische Bibliotheken: Archive von Objekt-Dateien. Name: lib<name>.a, referenziert wird nur <name>. Linker behandelt statische Bibliotheken wie mehrere Objekt-Dateien. Ursprünglich gibt es nur statische Bibliotheken (Einfach zu implementieren, aber funktionalitätlos).

Dynamische Bibliotheken: Linken erst zur Laufzeit bzw. Laufzeit des Programms. Höherer Aufwand, jedoch austauschbar. Executable 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. mode ist einer 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 werden nicht für andere Objektdateien verwendet

void *dlsym (void *handle, char *name): gibt die Adresse des Symbols name aus der mit handle bezeichneten Bibliothek zurück. Neue Typinformationen (Variable? Funktion?) Wenn keine Informationen gefunden werden, wird ein nullpointer zurückgegeben. Kann zu undefined behaviour führen.

```
// type "func_t" is a address of a function with a int param and int return type
typedef bool (*func_t)(bool);
handle = dlopen("libc.so.1", RTLD_NOW); // open library
func_t function = dlsym(handle, "mycoolfunction"); // write mycoolfunction addr. into a func_t
bool *b = dlsym(handle, "mycoolbool"); // get address of "mycoolbool"
(*function)(b); // call "mycoolfunction" with "mycoolbool" 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 + Bibliotheksname + .so (z.B. libmylib.so)

SO-Name: Linker-Name + . + Versionsnummer (z.B. libmylib.so.2)

Real-Name: SO-Name + . + Interventionsnummer (z.B. libmylib.so.2.2)

Shared Objects: Nahezu alle Executables benötigen zwei Shared Objects: *libc.so:* Standard C Library, *ld.so:* Shared Object loader (Lib Shared Objects loader).

Implementierung dynamischer Bibliotheken: Müssen verschreibbar sein, mehrere müssen in den gleichen Prozess geladen werden. Die Aufgabe des Linkers wird in den Loader bzw. Dynamic Linker verschoben (Load Relocation).

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 gemappt. Benötigt Position-Independent-Code (Adressen nur relativ zum instruction Pointer, Prozessor muss relative Instruktionen anordnen).

Relative Moves via Relative Calls: Mittels Hilfsfunktion wird Rückzugsadresse in Register abgelegt, somit kann relativ dazu gearbeitet werden.

Global Offset Table (GOT): Pro dynamische Bibliothek & Executable vorhanden, enthält pro Symbol einen Eintrag. Der Loader füllt zur Laufzeit die Adressen in die GOT ein.

Procedure Lookup Table (PLT): implementiert Lazy Binding. Enthält pro Funktion einen Eintrag, dieser enthält Sprungbehlief an Adresse in GOT-Eintrag. Dieser zeigt auf eine Proxy-Funktion, welche den GOT-Eintrag überschreibt. Vorteil: erspart bedingten Sprung.

5. THREADS

Jeder Prozess hat virtuell den ganzen Rechner für sich alleine. Prozesse sind gut geeignet für unabhängige Applikationen. Nachteile: Realisierung paralleler Abläufe innerhalb derselben Applikation ist aufwändig, Overhead zu gross falls nur kürzere Teilaktivitäten, gemeinsame Ressourcennutzung ist erschwert.

Threads: parallel ablaufende Aktivitäten innerhalb eines Prozesses, welche auf alle Ressourcen im Prozess gleichermassen Zugriff haben. Benötigen eigenen Kontext und eigenen Stack. Informationen werden in einem Thread-Control-Block abgelegt.

5.1. AMDAHL'S REGEL

Nur bestimmte Teile eines Algorithmus können parallelisiert werden.

T₁ Ausführungzeit, wenn komplett serial durchgeführt (im Bild $T_1 = T_2 + T_3 + T_4 + T_5$)

T₂ Ausführungzeit, wenn maximal parallelisierte gewisse Schritte

T₃ Ausführungzeit für den Anteil, der serial ausgeführt werden muss (im Bild $T_3 = T_2 + T_4 + T_5$)

(T - T₃)/n Parallel-Anteil verteilt auf alle n Prozessoren. (im Bild: $(T_1 - T_3)/n$)

T₁ + T₃/n Serialer Teil + Paralleler Teil = T'
Die serielle Variante benötigt also höchstens f mal mehr Zeit als die parallele Variante:

$$f \leq \frac{T}{T'} = \frac{T}{T_1 + \frac{T_3}{n}}$$

f heisst auch Speedup-Faktor, weil die parallele Variante max. f-mal schneller ist als die serielle.

Definiert man $s = T_1/T_3$, also den seriellen Anteil am Algorithmus, dann ist $s = T' = T_1$. Dadurch erhält man f unabhängig von der Zeit:

$$f \leq \frac{T}{T'} = \frac{T}{T_1 + \frac{T_3}{n}} = \frac{T}{s + T_1 + \frac{T_3}{n}} \Rightarrow f \leq \frac{1}{\frac{1}{s} + \frac{1}{n}}$$

5.1.1. Bedeutung

Abbildung einer oberen Schranke für den maximalen Geschwindigkeitsgewinn

Nur wenn alles parallelisierbar ist, ist der Speedup proportional und maximal f(n, n) = n

Sonst ist der Speedup mit höherer Prozessor-Anzahl immer geringer (Kurve flach ab)

(1/n, n) rein seriell

Grenzwert:

Mit höherer Anzahl Prozessoren nähert sich der Speedup 1 an:

$$\lim_{n \rightarrow \infty} \frac{1-s}{n} = 0 \quad \lim_{n \rightarrow \infty} s + \frac{1-s}{n} = s \quad \lim_{n \rightarrow \infty} \frac{1}{\frac{1}{s} + \frac{1}{n}} = \frac{1}{\frac{1}{s}} = s$$

5.2. POSIX THREAD API

```
int pthread_create(
    pthread_t *thread_id, pthread_attr_t const *attributes,
    void * (*start_function) (void *), void *argument)
```

erzeugt einen Thread, die ID des neuen Threads wird im Out-Parameter thread_id zurückgegeben. attributes ist ein opakes Objekt, mit dem z.B. die Stack-Größe spezifiziert werden kann. Die erste auszuführende Instruktion ist die Funktion in start_function. argument ist ein Pointer auf eine Datenstruktur auf dem Heap für die Argumente für start_function.

```
// Erstellung
struct T { // params of function
    int value;
};
// Verwendung
void start_my_thread (void) {
    struct T t = malloc(sizeof(struct T));
    t->value = 100; // set argument
    pthread_t tid;
    pthread_create (&tid, NULL, start_my_thread, (void *)t);
    // free arg: t = p; arg;
    printf ("tid=%d, p=>value");
    return 0;
};
```

Thread-Attribute: **pthread_attr_t attr;** // Variabel erstellen **pthread_attr_t init (&attr);** // Variabel initialisieren **pthread_attr_t setattrsize (&attr, 1 <= 16);** // 64kb Stackgröße **pthread_create (&tid, &attr, ...);** // Thread erstellen **pthread_attr_t destroy (&attr);** // Attribute löschen

Lebensdauer: Jede solange, bis er aus der Funktion start_function zurückspringt, er pthread_exit oder ein anderer Thread pthread_cancel aufruft oder sein Prozess beendet wird.

pthread_exit (void *return_value): Beendet den Thread und gibt den return_value zurück. Das ist äquivalent zum Rücksprung aus start_function mit dem Rückgabewert.

int pthread_cancel (pthread_t thread_id): Sendet eine Anforderung, dass der Thread mit thread_id beendet werden soll. Die Funktion wartet nicht, dass der Thread tatsächlich beendet wurde. Der Rückgabewert ist 0, wenn der Thread existiert, bzw. ESRCH (error, such), wenn nicht. **int pthread_detach (pthread_t thread_id):** Entfernt den Speicher, den ein Thread belegt hat, falls dieser bereits beendet wurde. Beendet den Thread aber nicht. (*Itzschl's Daemon Thread*) **int pthread_join (pthread_t thread_id, void **return_value):** Wartet

