Übungen Betriebssysteme (BS)

U5 – Dateioperationen

Alwin Berger

TU Dortmund - AG Systemsoftware

Agenda

- Besprechung von “A4 - Speicherverwaltung”
- Vorbesprechung zu “A5 – Dateioperationen”
  - Dateioperationen in C
  - Eigenschaften von Dateien
  - Verzeichnisse lesen

Dateioperationen in C

- Exemplarischer Ablauf
  1. Datei öffnen
  2. Lesen
  3. Schreiben
  4. …
  5. Datei schließen

Dateioperationen in C

- Exemplarischer Ablauf
  1. Datei öffnen
  2. Lesen
  3. Schreiben
  4. …
  5. Datei schließen

Dateioperationen in C

- Exemplarischer Ablauf
  1. Datei öffnen
  2. Lesen
  3. Schreiben
  4. …
  5. Datei schließen
- Öffnen einer Datei liefert Handle
  - Wird bei jeder Operation gebraucht - inkl. Schließen
  - Vermerkt den Modus (Lesen, Schreiben, beides) der geöffneten Datei
  - Beinhaltet Schreib-Lese-Zeiger

Dateioperationen in C

- Exemplarischer Ablauf
  1. Datei öffnen
  2. Lesen
  3. Schreiben
  4. …
  5. Datei schließen
- Öffnen einer Datei liefert Handle
  - Wird bei jeder Operation gebraucht - inkl. Schließen
  - Vermerkt den Modus (Lesen, Schreiben, beides) der geöffneten Datei
  - Beinhaltet Schreib-Lese-Zeiger
- eine Datei kann mehrfach geöffnet werden

Dateioperationen in C

- Exemplarischer Ablauf
  1. Datei öffnen
  2. Lesen
  3. Schreiben
  4. ...
  5. Datei schließen
- Öffnen einer Datei liefert Handle
  - Wird bei jeder Operation gebraucht - inkl. Schließen
  - Vermerkt den Modus (Lesen, Schreiben, beides) der geöffneten Datei
  - Beinhaltet Schreib-Lese-Zeiger
- eine Datei kann mehrfach geöffnet werden
- Dateien müssen **nach** ihrer Verwendung geschlossen werden
  → Vermeidung von Ressourcenlecks

Schnittstellen für Dateioperationen in C

- Unter Linux gibt es mehrere Möglichkeiten

Schnittstellen für Dateioperationen in C

- Unter Linux gibt es mehrere Möglichkeiten
  - Syscall-Wrapper der C-Bibliothek (“low-level”), z.B. open(2)

```
int open(const char *path, int flags)
  syscall(__NR_open, path, flags)
```

Schnittstellen für Dateioperationen in C

- Unter Linux gibt es mehrere Möglichkeiten
  - Syscall-Wrapper der C-Bibliothek (“low-level”), z.B. `open(2)`

```
int open(const char *path, int flags)
  syscall(__NR_open, path, flags)
```

  - Abstrakte Stream-Schnittstelle in C (“high-level”), z.B. `fopen(3)`

```
FILE* fopen(const char *path, const char *mode)
  Abstraktion von Filedeskriptor zu Stream (FILE*)
    open(path, flags)
```

“low-level”-Dateioperationen

- open(2): Datei öffnen
  - `int open(const char *path, int flags)`
  - Gibt Dateideskriptor zurück - wird später benötigt
- read(2): Aus Datei lesen
  - `size_t read(int fd, void *buf, size_t count)`
- lseek(2): Schreib-/Leseposition verändern
  - `off_t lseek(int fd, off_t offset, int whence)`
- close(2): Offene Datei schließen
  - `int close(int fd)`
- Standardisierte Schnittstelle, u.a. in POSIX.1-2001

“low-level”-Beispiel

```c
#define READ_SIZE 64
char buf[READ_SIZE+1];

int main(int argc, char *argv[]) {
  int fd = 0;

  fd = open("somefile", O_RDONLY);
  if (fd == -1) {
    /* ... Fehler ... */
  }

  if (read(fd, buf, READ_SIZE) < READ_SIZE) {
    /* Die Datei ist kleiner als 64 Byte */
  }

  buf[READ_SIZE] = 0;
  printf("Die ersten 64 Zeichen sind: %s\n", buf);

  if (lseek(fd, 1000, SEEK_SET) != 1000) {
    /* ... Fehler ... */
  }
```

```c
  if (read(fd, buf, 1) < 1) {
    /* ... Fehler ... */
  }

  printf("An Position 1000 steht das Zeichen %c\n", buf[0]);

  if (close(fd) == -1) {
    /* ... Fehler ... */
  }
}
```

C-Streams (“high-level”)

- Abstrakter, plattformübergreifender Kommunikationskanal
- Teil des C-Standards!
- Interne Pufferung: Blockweises Lesen → höhere Verarbeitungsgeschwindigkeit
- Unabhängiges Lesen und Schreiben an verschiedenen Positionen
- Stream-Repräsentation durch `struct FILE`
- Definierte Standard-Streams: `stdin, stdout, stderr`

“high-level”-Dateioperationen (1)

- fopen(3): Datei öffnen
  - `FILE* fopen(const char *path, const char *mode)`
  - `mode`: z.B. “r” (Lesen), “r+” (Schreiben & Lesen), “a” (Anhängen)
    - Gibt Zeiger auf `FILE`-Datenstruktur zurück - wird später benötigt
- fread(3): Aus Datei lesen
  - `size_t fread(void *buf, size_t itemsize, size_t count, FILE *stream)`
- fseek(3): Schreib-/Leseposition verändern
  - `off_t fseek(FILE *stream, off_t offset, int whence)`
- fclose(3): Offene Datei schließen
  - `int fclose(FILE *stream)`
- Ebenfalls standardisierte Schnittstelle, u.a. in POSIX.1-2001

“high-level”-Beispiel

```c
#define READ_SIZE 64
char buf[READ_SIZE+1];

int main(int argc, char *argv[]) {
  FILE *somefile;

  somefile = fopen("somefile", "r");
  if (somefile == NULL) {
    /* ... Fehler ... */
  }

  if (fread(buf, sizeof(*buf), READ_SIZE, somefile) < READ_SIZE) {
    /* Die Datei ist kleiner als 64 Byte */
  }

  buf[READ_SIZE] = 0;
  printf("Die ersten 64 Zeichen sind: %s\n", buf);

  if (fseek(somefile, 1000, SEEK_SET) == -1) {
    /* ... Fehler ... */
  }
```

```c
  if (fread(buf, sizeof(*buf), 1, somefile) < 1) {
    /* ... Fehler ... */
  }

  printf("An Position 1000 steht das Zeichen %c\n", buf

  if (fclose(somefile) == -1) {
    /* ... Fehler ... */
  }
}
```

“high-level”-Dateioperationen (2)

- ftell(3): Gibt den Datei-Positionszeiger für den Stream `stream` aus
  - `long ftell(FILE *stream)`
- fscanf(3): Liest den Inhalt einer Datei gemäß Formatstring aus
  - `int fscanf(FILE *stream, const char *format, ...)`
  - Funktioniert wie `scanf(3)`
- fprintf(3): Schreibt einen formatierten String in eine Datei
  - `int fprintf(FILE *stream, const char *format, ...)`
  - Analog zu `printf(3)`

Dateiinformationen abfragen

- Linux bzw. Unix bildet (fast) alles auf Dateien ab: Geräte, Sockets, …
- Dateien haben verschiedene Eigenschaften
  - Typ: Verzeichnis, Datei, zeichen- oder blockorientiertes Gerät
  - Eigentümer:in (Nutzer:in und Gruppe)
  - Größe
  - Berechtigungen

Dateiinformationen abfragen

- Linux bzw. Unix bildet (fast) alles auf Dateien ab: Geräte, Sockets, …
- Dateien haben verschiedene Eigenschaften
  - Typ: Verzeichnis, Datei, zeichen- oder blockorientiertes Gerät
  - Eigentümer:in (Nutzer:in und Gruppe)
  - Größe
  - Berechtigungen
- Abfragen dieser Eigenschaften mittels `stat(2)`
  - `int fstat(const char *pathname, struct stat *finfo)`
  - `pathname`: Name der Datei
  - `finfo`: Zeiger auf Datenstruktur der Dateieigenschaften

`stat`-Beispiel

- Vordefinierte Makros, um Typ anhand von `st_mode` zu ermitteln (siehe `man 7 inode`)
  - `S_ISREG(st_mode)` → reguläre Datei
  - `S_ISDIR(st_mode)` → Verzeichnis
- `st_size` liefert die Dateigröße in Bytes
- Weitere Informationen in `man 2 stat`

```c
int main(int argc, char *argv[]) {
  struct stat finfo;

  for(int i = 1; i < argc; i++) {
    if (stat(argv[i], &finfo) == -1) {
      /* Fehlerbehandlung */
    }
    if (S_ISREG(finfo.st_mode)) {
      printf("%s ist eine %d Byte große Datei\n", argv[i], finfo.st_size);
    } else if (S_ISDIR(finfo.st_mode)) {
      /* Verzeichnis */
    } else { /* z.B. FIFO, Gerät, ... */}
  }
}
```

Verzeichnisoperationen

- opendir(3): Öffnet ein Verzeichnis zum Lesen
  - `DIR* opendir(const char *path)`
  - Liefert einen Zeiger auf `DIR`-Datenstruktur - wird später benötigt
- readdir(3): Liest den nächsten Eintrag aus dem Verzeichnis
  - `struct dirent* opendir(DIR *dir)`
  - Liefer einen Zeiger auf `dirent`-Datenstruktur
  - Gibt Auskunft über den aktuellen Eintrag im Verzeichnis
  - Siehe `man 3 readdir`
- closedir(3): Schließt das aktuelle Verzeichnis
  - `int closedir(DIR *dir)`

Beispiel zu Verzeichnisoperationen

```c
#include <sys/types.h>
#include <dirent.h>
#include <stdio.h>

int main(int argc , char *argv[]) {
  int retval = 0, count = 0;
  DIR *dir = NULL;
  struct dirent *entry = NULL;

  /* Verzeichnis öffnen
     DIR-Zeiger erstellen */
  dir = opendir("/tmp");
  if (!dir) {
    /* ... Fehler ... */
  }

  /* jeder Aufruf liefert einen Eintrag
     NULL -> Listenende */
  while ((entry = readdir(dir))) {
    printf ("Eintrag %d: %s\n", ++count , entry->d_name);
  }
  printf ("Anzahl: %d\n", count);

  retval = closedir(dir);
  if (retval == -1) {
    /* ... Fehler ... */
  }

  return 0;
}
```

Dateipfade in C erstellen

- Speicherplatz für zu konstruierenden Pfad sicherstellen
- Benötigter Speicher hängt von der Länge ab
  - “foo/bar.txt” vs. “irgendein/furchtbar/langer/Pfad.txt”
- Zur Erinnerung:
  - Länge einer Zeichenkette mit `strlen(3)` bestimmen
  - Speicher mit `malloc(3)` allozieren
- **Wichtig:** Speicher abschließend freigeben (`free(3)`)

Dateipfade in C erstellen

- Speicherplatz für zu konstruierenden Pfad sicherstellen
- Benötigter Speicher hängt von der Länge ab
  - “foo/bar.txt” vs. “irgendein/furchtbar/langer/Pfad.txt”
- Zur Erinnerung:
  - Länge einer Zeichenkette mit `strlen(3)` bestimmen
  - Speicher mit `malloc(3)` allozieren
- **Wichtig:** Speicher abschließend freigeben (`free(3)`)

```c
int bufsize = strlen("foo") + strlen("bar") + 2;

char *buf = malloc(bufsize);
if (buf == NULL) { /* Fehler */ }

snprintf (buf, bufsize, "%s/%s", "foo", "bar");
// buf enthält jetzt "foo/bar"
// (inkl. terminierendem Nullbyte)
```

Dateipfade in C erstellen

- Speicherplatz für zu konstruierenden Pfad sicherstellen
- Benötigter Speicher hängt von der Länge ab
  - “foo/bar.txt” vs. “irgendein/furchtbar/langer/Pfad.txt”
- Zur Erinnerung:
  - Länge einer Zeichenkette mit `strlen(3)` bestimmen
  - Speicher mit `malloc(3)` allozieren
- **Wichtig:** Speicher abschließend freigeben (`free(3)`)

```c
int bufsize = strlen("foo") + strlen("bar") + 2;

char *buf = malloc(bufsize);
if (buf == NULL) { /* Fehler */ }

snprintf (buf, bufsize, "%s/%s", "foo", "bar");
// buf enthält jetzt "foo/bar"
// (inkl. terminierendem Nullbyte)
```

← Warum 2 Bytes zusätzlich?

Hilfreiche Stringfunktionen

- Notwendiger Header: `string.h`
- Kopieren eines Strings mittels
  `char* strncpy(char *to, const char *from, size_t count)`
- Konkatenieren zweier Strings mit
  `char* strncat(char *str1, const char *str2, size_t count)`

Was haben wir heute gelernt?

- Zusammenfassung
  - Dateioperationen in C
  - Verzeichnisse lesen