SP-Codeschnipsel Übersicht

Standardfehler:

- **`SIZEOF - sizeof(char);`**
- Argc immer prüfen, auch auf 1!!
- **`static`** `bei private Methoden und variablen`
- Wenn eine Funktion keine Parameter hat, dann nicht als *int f() {};* deklarieren, sondern als
  - `int f(void) {};`
- `if(fprintf(...) < 0) perror("fprintf"); //Fehlerbehandlung!`
- `fgets returns NULL`
- `fgetc/fputc return EOF`
- `fopen(char *pathname, ...) / fdopen(int fd, ...), fclose(FILE *f)`
- *fflush* von *stdout* nicht vergessen
- Die Fehlerbehandlung kann in der Klausur bei folgenden Funktionen weggelassen werden:
  - `pthread_mutex_{lock,unlock,destroy}`
  - `pthread_cond_{signal,broadcast,wait,destroy}`
  - `sigaction (falls die Parameter fix sind, also keine neuen Fehler zur Laufzeit auftreten können)`
  - `sigprocmask`
  - `sigsuspend`

Wichtige Header

```
#include <dirent.h>
#include <errno.h>
#include <fcntl.h>
#include <limits.h>
#include <pthread.h>
#include <signal.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
```

String-Manipulation

```
int strcmp(const char *s1, const char *s2);
// returns 0 if equal, kann nicht fehlschlagen
   if (!strcmp(argv[1], "-v")) {
      //do sth
   }

int strncmp(const char *s1, const char *s2, size_t n);
// wie strcmp, vergleicht aber nur die ersten n Zeichen

size_t strlen(const char *s);
// gibt Länge des String zurück(ohne \0), klappt immer

char* strdup(const char *s):
// allokiert intern Speicher (mittels malloc) und schreibt da die Kopie
rein, bei Fehler Null und setzt errno

char *strcpy(char *dest, const char *src);
// Kopiert src in dest, kann nicht fehlschlagen:
   char cmdCpy[strlen(cmdline) +1];
   strcpy(cmdCpy, cmdline);

char *strcat(char *restrict dst, const char *restrict src);
// Hängt src an dst an (ab dem \0 von dst)
     char full_path[strlen(path) + strlen(address) + 1];
     strcpy(full_path, path);
     strcat(full_path, address);

int sprintf(char *restrict str, const char *restrict format, ...);
// Wie fprintf nur wird die Ausgabe in str geschrieben (str muss passend
groß sein)
   char buf[n];
   if (sprintf(buf, "Hi %s", a) < 0) {
       perror("sprintf");
   }

char *strchr(const char *s, int c);
// gibt Pointer zum ersten Auftauchen von c in s zurück, wenn nicht gefunden
NULL
   if (strchr(command, '\n') != NULL) { // do something}
```

strtok(char* str, const char* delim)
// bekommt str und Liste möglicher Trennzeichen(delim). Zerteilt str in einzelne Tokens, die mit Trennzeichen getrennt sind. Beim ersten Aufruf str mitgeben, danach NULL um weiter auf dem selben String zu bleiben. Gibt jedes Mal Pointer zum nächsten Token zurück, am Ende NULL. Zerlegt String dabei also wenn man ihn behalten will kopieren.

```
char input[] = "as dfs d fsd erd";
char *args[strlen(input) / 2 + 1];
args[0] = strtok(input, " ");
int index = 1;
while ((args[index++] = strtok(NULL, " ")) != NULL);
```

strtok_r(char* str, const char* delim, char saveptr);**
// wie strtok aber auch thread safe man muss nur saveptr auf den Stack und mit &saveptr aufrufen

```
char *saveptr;
strtok_r(input, " ", &saveptr);
```

long strtol(const char *restrict nptr, char **restrict endptr, int base);
// Parsed den String nptr in einen long zu einem gegebenen Basis-System (base = 10, für Dezimalsystem). Wenn der String noch andere Zeichen enthält, wird das erste fehlehafte Char in endtr gespeichert.

```
errno = 0;
char *endptr;
long x = strtol(str, &endptr, 10);
if (errno != 0) {
    die("strtol");
}
if (str == endptr || *endptr != '\0') {
    fprintf(stderr, "invalid number\n");
    exit(EXIT_FAILURE);
}
```

Multi-Threading

pid_t fork(void);

```
pid_t p = fork();
if (p == -1) {
   die("fork");
} else if (p == 0) {
   //Kindprozess
} else {
```

```
      //Elternprozess, p ist die PID des neuen Kindprozesses
   }
```

pid_t wait(int *wstatus);

```
   int status;
   if (wait(&status) == -1) {
     perror("wait");
   }
```

pid_t waitpid(pid_t pid, int *wstatus, int options);

```
   int status;
   pid_t child = waitpid(-1, &status, WNOHANG);
   if (child == -1) {
     perror("waitpid");
   } else if (child == 0){
     //childs havent terminated yet
   } else {
     //do sth
```

Status ausgeben

```
     if (!WIFEXITED(status)) { // Kind mit Fehler beendet
        if(fprintf(stderr, "%i\n", WEXITSTATUS(status)) < 0){
            perror("fprintf");
        }
     }
   }
```

Auf Beendigung eines <u>bestimmten</u> childs warten

```
   pid_t pid = fork();
   ... // im parent process
   int wstatus;
   while (waitpid(pid, &wstatus, 0) != -1) {
       if (WIFEXITED(wstatus) || WIFSIGNALED(wstatus)) {
           return;
       }
   }
   perror("waitpid");
```

int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);

```
   static void *worker(void *args) {
       return NULL;  // muss einen void * returnen
   }
   ...
```

```
  pthread_t pthread; // Falls nachher gejoined werden muss, pthread_t[n]
  Array erstellen
  int tmp_errno = pthread_create(&pthread, NULL, &worker, args);
  if (tmp_errno != 0) {
      errno = tmp_errno;
      die("pthread_create");
  }
```

int pthread_join(pthread_t thread, void **retval);

```
  int tmp_errno = pthread_join(&thread));
  if (tmp_errno != 0) {
      errno = tmp_errno;
      die("pthread_join");
  }
```

int pthread_detach(pthread_t thread);

```
  int tmp_errno = pthread_detach(pthread_self());
  if (tmp_errno != 0) {
      errno = tmp_errno;
      perror("pthread_detach"); // nicht fatal
  }
```

Synchronisation

Atomic

```
  #include<stdatomic.h> // atomic_int , atomic_long
  atomic_int counter = ATOMIC_VAR_INIT(5); // int counter = 5;
  int d = atomic_load(&counter);           // int d = counter;
  atomic_store(&counter, 10);              // counter = 10
  atomic_fetch_add(&counter, 1);           // counter++
```

CAS (Nicht-blockierende Synchronisation)

```
  atomic_int a = ATOMIC_VAR_INIT(10);
  int old, local;
  do {
      old = atomic_load(&a);
      local = old * 2;
  } while (!atomic_compare_exchange_strong(&a, &old, local));
```

Mutex (Semaphore)

```
  static volatile int counter; // wegen den Mutexen muss der counter kein
  Atomic sein
  static pthread_mutex_t m;
  static pthread_cond_t c;

  void P() {
      pthread_mutex_lock(&m);
      while (counter <= 0) {
          pthread_cond_wait(&c, &m);
      }
      counter--;
      pthread_mutex_unlock(&m);
  }

  void V() {
      pthread_mutex_lock(&m);
      counter++;
      pthread_cond_signal(&c);
      pthread_mutex_unlock(&m);
  }
```

Exec

```
int execvp(const char *file, char *const argv[]);
int execlp(const char *file, const char *arg, ..., NULL);
  execvp(args[0], args);
  perror("execvp");  //kehrt nur im Fehlerfall zurück

int fcntl(int fd, int cmd, ... /* arg */ );
  int flags = fcntl(fd, F_GETFD, 0);
  if (flags == -1) {
      die("fcntl");
  }
  if (fcntl(fd, F_SETFD, flags | FD_CLOEXEC) == -1) {
      die("fcntl");
  }
```

Filestreams

```
int open(const char *pathname, int flags);
  TODO,  int file_fd = open(full_path, O_RDONLY | O_CLOEXEC);
```

```
int creat(const char *pathname, mode_t mode);
   TODO

FILE* fdopen(int filedes, const char *mode):
// gibt File* fuer das den filedes (bspw. ein Socket) zurueck
// mode bspw. "r" (read) oder "w" (write) oder "w+" (beides)
   FILE* rx = fdopen(clientSock, "r");
   if(rx == NULL){
       die("fdopen");
   }

FILE *fopen(const char* pathname, const char* mode);
//öffnet pathname mit Modus mode(bspw. "r" für reading, "r+" für read and
write)
//gibt NULL im Fehlerfall zurück und setzt errno
   FILE *file = fopen("path", "r");
   if (file == NULL) {
      die("fopen");
   }

int fclose(FILE* stream);
// flushes stream und closes the file
// gibt 0 bei Erfolg zurück, sont EOF und setzt errno
   if (fclose(file) == EOF) {
       die("fclose");
   }

int close(int filedes); // wie fclose fuer filedeskriptor
   if (close(listenSock) == -1) {
       die("fclose");
   }

int fileno(FILE *stream);
// returnt die Filedescriptor-Zahl

int fputc(int c, FILE *stream);
   if (fputc('c', file) == EOF) {
      perror("fputc");
   }

int fputs(const char* s, FILE *stream);
```

```
// schreibt s in den stream ohne \0, gibt bei Erfolg nonnegative number und
bei Fehler EOF zurück, errno wird nicht gesetzt
//Code siehe fputc

int fgetc(FILE *stream);
// liest nächsten  char von stream und gibt ihn als int zurück, oder EOF bei
end of file, aber auch im Fehlerfall
   int c;
   while ((c = fgetc(stream)) != EOF) {}
   if (ferror(stream)) {
       die("fgetc");
   }


char *fgets(char *s, int size, FILE *stream);
// gibt NULL im Fehlerfall und bei end of file zurück
// liest Zeichen von Dateikanal fp in das Feld s bis entweder size-1 Zeichen
gelesen wurden oder \n gelesen oder EOF erreicht wurde
// s wird mit \0 abgeschlossen (\n wird nicht entfernt)
   char buf[n];
   while (fgets(buf, sizeof(buf), stream) != NULL) {
       size_t len = strlen(buf);
       if (len > sizeof(buf) - 2 && buf[len - 1] != '\n') {
           int c; // Überlange Zeile -> Alles weglesen
           while ((c = fgetc(stream)) != EOF) {
               if (c == '\n') {
                   break;
               }
           }
           if (ferror(stream)) {
               die("fgetc");
           }
           continue; // Zeile ignorieren
       }

       if (buf[len - 1] == '\n') { // kann evtl. auch mit strtok gemacht
   werden
           buf[len - 1] = '\0';
       }
       // Zeile verarbeiten
   }
   if (ferror(stream)) {
       die("fgets");
```

```
   }

int fstat(int fd, struct stat *statbuf);
   int file_fd;
   struct stat statbuf;
   if (fstat(file_fd, &statbuf) == -1) {
       die("fstat");
   }
   if (S_ISDIR(statbuf.st_mode)) { } // directory
   else if (S_ISREG(statbuf.st_mode)) { } // file
   else { } // error

int lstat(const char *pathname, struct stat *statbuf);
// TODO

int dup(int oldfd);
// sehr oft rx und tx aus einem Filedeskriptor fd
// angenommen man soll nicht sterben sondern nur alle Ressourcen freigeben
und -1 returnen
   FILE *rx = fdopen(fd, "r");
   if(!rx){
        perror("fdopen");
        close(fd);
        return -1;
   }
   int fd2 = dup(fd);
   if(fd2 == -1){
       perror("dup");
       fclose(rx);
       return -1;
   }
   FILE *tx = fdopen(fd2, "w");
   if(!tx){
        perror("fdopen");
        fclose(rx);
        close(fd2);
        return -1;
   }

int dup2(int fd, int newfd);
// Schließt newfd und dupliziert fd auf den Wert von newfd, damit lässt sich
bspw. stdout in eine Datei umleiten
   int fd = creat("/dev/null", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
```

```
  if (fd == -1){
      die("open");
  }
  if (fflush(stdout) == EOF) {
      die("fflush");
  }
  if (dup2(fd, STDOUT_FILENO) == -1) {
      die("dup2");
  }
  if (close(fd) == -1){
      perror("close");
  }
```

Directories

DIR *opendir(const char *name);

```
  DIR *dir = opendir("path");
  if (dir == NULL) {
      perror("opendir");
  }
```

struct dirent *readdir(DIR *dirp);

```
  struct dirent *dirent;
  while (errno = 0, (dirent = readdir(dir)) != NULL){
    //do sth
  }
  if (errno) {
    perror("readdir");  //evtl. noch closedir
  }
```

int closedir(DIR *dirp);

```
  if (closedir(dir) == -1) {
    perror("closedir");
  }
```

Ein ganzes Directory durchgehen:

```
  char *path = ".";
  DIR *dir = opendir(path);
  if (dir == NULL) {
      die("opendir");
  }
  struct dirent *dirent;
```

```
  while (errno = 0, (dirent = readdir(dir)) != NULL) {
      if (strcmp(dirent->d_name, ".") == 0 ||
          strcmp(dirent->d_name, "..") == 0) {
          continue;
      }
      // aktuellen Pfad zusammenbauen
      char npath[strlen(path) + strlen(dirent->d_name) + 2];
      if (sprintf(npath, "%s/%s", path, dirent->d_name) < 0) {
          perror("sprintf");
          continue;
      }
      struct stat info;
      if (lstat(npath, &info) == -1) {
          perror("lstat");
          continue;
      }
      if (S_ISREG(info.st_mode)) {
          // regular file
      } else if (S_ISDIR(info.st_mode)) {
          // directory
      }
  }
  if (errno) {
      die("readdir");
  }
  if (closedir(dir) == -1) {
      die("closedir");
  }
```

int scandir(const char *dirp, struct dirent ***namelist, int (*filter)(const struct dirent *), int (*compar)(const struct dirent **, const struct dirent **));

int alphasort(const struct dirent **a, const struct dirent **b)

```
  static int dir_filter(const struct dirent *dirent) {
      return dirent->d_name[0] != '.'; // exclude all dirs starting with .
  }
  ...
  struct dirent **entries;
  int size = scandir(path, &entries, &dir_filter, &alphasort);
  if (size == -1) {
      die("scandir");
  }
```

Signale

Signal-Handler mit sigaction

```
struct sigaction action = {
    .sa_handler = &signal_handler, // SIG_IGN, SIG_DFL
    .sa_flags = SA_RESTART,
};
sigemptyset(&action.sa_mask);
sigaction(SIGNAL, &action, NULL);
...
// signal handler koennen nicht unterbrochen werden
static void signal_handler(int signal) {
    if (signal != SIGNAL) {
        return;
    }
    int tmp_errno = errno
    ... // kein perror oder fprintf verwenden!! (s. unten)
    errno = tmp_errno;
}
```

Fehlerausgabe auf stderr im Signal-Handler

(perror & fprintf verwenden interne Buffer und dürfen deswegen nicht verwendet werden)

```
char *error = "fehler";
write(STDERR_FILENO, error, strlen(error));
```

Signal blockieren

```
(static volatile int event; // im Signal-Handler auf 1 setzen)
...
event = 0;
sigset_t oldmask, mask;
sigemptyset(&mask);
sigaddset(&mask, SIGNAL); // SIGCHLD oderso
sigprocmask(SIG_BLOCK, &mask, &oldmask);
```

Evtl. dazwischen auf Signal warten:

```
while (event == 0) {
    sigsuspend(&oldmask);
}
event = 0;
```

Signal deblockieren

```
sigprocmask(SIG_SETMASK, &oldmask, NULL);
```

Sockets

Server-Socket

```
struct sigaction action = {
    .sa_handler = SIG_IGN,
};
sigemptyset(&action.sa_mask);
sigaction(SIGPIPE, &action, NULL);

int server_sock = socket(AF_INET6, SOCK_STREAM, 0);
if (server_sock == -1) {
    die("socket");
}
// Optional um Port nicht temporär zu blockieren, wenn Server beendet
wurde
int flag = 1;
if (setsockopt(server_sock, SOL_SOCKET, SO_REUSEADDR, &flag,
sizeof(flag)) == -1) {
    die("setsockopt");
}
struct sockaddr_in6 name = {
    .sin6_family = AF_INET6,
    .sin6_port = htons(port), // Port-Nummer
    .sin6_addr = in6addr_any, // IPv6-Adresse
};
if (bind(server_sock, (struct sockaddr *)&name, sizeof(name)) == -1) {
    die("bind");
}
if (listen(server_sock, SOMAXCONN) == -1) {
    die("listen");
}
while (1) {
    int client_sock = accept(server_sock, NULL, NULL);
    if (client_sock == -1) {
        perror("accept"); // Kein fataler Fehler
    } else{
        handleConnection(client_sock, server_sock);
    }
}
```

Client-Socket (DNS Anfrage und verbinden mit URL)

```
char *url;
```

```
char *port;
struct addrinfo hints = {
    .ai_socktype = SOCK_STREAM,
    .ai_family = AF_UNSPEC, // Either IPv4 or IPv6
    .ai_flags = AI_ADDRCONFIG,
};
struct addrinfo *head;
ret_addrinfo = getaddrinfo(url, port, &hints, &head); // 25 für smtp
if (ret_addrinfo == EAI_SYSTEM) {
    die("getaddrinfo");
} else if (ret_addrinfo != 0) {
    fprintf(stderr, "getaddrinfo error: %s\n",
gai_strerror(ret_addrinfo));
    exit(EXIT_FAILURE);
}

int sock;
struct addrinfo *curr;
for (curr = head; curr != NULL; curr = curr->ai_next) {
    sock = socket(curr->ai_family, curr->ai_socktype, curr->ai_protocol);
    if (sock == -1) {
        perror("socket"); // kein fataler Fehler
        continue;
    }

    int ret_con = connect(sock, curr->ai_addr, curr->ai_addrlen);
    if (ret_con == -1) {
        perror("connect"); // kein fataler Fehler
    } else if (ret_con == 0) {
        break;
    } else {
        if (close(sock) == -1) {
            perror("close"); // kein fataler Fehler
        }
    }
}
if (curr == NULL) {
    fprintf(stderr, "did not find any useable address\n");
    exit(EXIT_FAILURE);
}
```

Kopien von anderen Aufgaben - TODO einsortieren

Benutzername herausfinden:

```
uid_t uid = getuid();
errno = 0;
struct passwd *pwuid = getpwuid(uid);
if (pwuid == NULL) {
    if (errno != 0) {
        die("getpwuid");
    } else {
        fprintf(stderr, "getpwuid did not found you, phantom\n");
        exit(EXIT_FAILURE);
    }
}
char *username_tmp = pwuid->pw_name;
char username[strlen(username_tmp) + 1];
strncpy(username, username_tmp, sizeof(username));
```

und vollen Anmeldenamen:

```
char *full_name_tmp = pwuid->pw_gecos;
char full_name_full[strlen(full_name_tmp) + 1];
strncpy(full_name_full, full_name_tmp, sizeof(full_name_full));
char *full_name = strtok(full_name_full, ",");
if (full_name == NULL) {
    fprintf(stderr, "pw_gecos is malformatted\n");
    exit(EXIT_FAILURE);
}
```

Status von waitpid überprüfen:

```
if (WIFEXITED(wstatus)) {
    printf("exited, status=%d\n", WEXITSTATUS(wstatus));
} else if (WIFSIGNALED(wstatus)) {
    printf("killed by signal %d\n", WTERMSIG(wstatus));
} else if (WIFSTOPPED(wstatus)) {
    printf("stopped by signal %d\n", WSTOPSIG(wstatus));
} else if (WIFCONTINUED(wstatus)) {
    printf("continued\n");
}
```

Auf Integer-Overflow prüfen

```
#include <limits.h>
if (xy > INT_MAX) ;
```

Shortcut für if(x == NULL) ;

```
if(!xy) ;
```