readdir, opendir:

```
* dir = opendir(path);
dir == NULL){
 die("opendir");

uct dirent *d;
while(1){
  errno=0;
  d=readdir(dir);
  if(d==NULL){
     break;
  }
  //ignore . und ..
  if(!strcmp(d->d_name,".") ||
      !strcmp(d->d_name,"..")){
     continue;
  }
  char newpath[strlen(path)+strlen(d->d_name)+2]
  if(sprintf(newpath,"%s/%s",path,d->d_name)<0){
     die("sprintf");
  }
  struct stat buf;
  if(lstat(newpath,&buf)){
     die("lstat"); // oder: if (errno==EACCESS){
  }                      continue;
  :                    }
}
if(errno!=0){
   die("readdir");
}
if(closedir(dir)==-1){
   die("closedir");
}
```

fnmatch:

```
ret=fnmatch(pattern,d->d_name,flags);
ret==FNM_NOMATCH){
 continue;
else if (ret!=0){
  return -1; (Fehler)
else{
 //match
  ...
```

- 0 wenn Suchmuster auch ohne führenden Punkt auf versteckte Dateien zutreffen soll (Nutzer muss Punkt nicht explizit angeben)
- FNM_PERIOD: führender Punkt in Pfad muss genau von Punkt in pattern getroffen werden
- ... FNM_PATHNAME: / müsste explizit angegeben

strtol (Strings -> Long konvertieren):

```
char *endptr;
char *ptr = argv[1]; (z.B)
errno=0;
long d=strtol(ptr,&endptr,10);     // 10: Zahlenbasis
if((endptr==ptr)||(*endptr!='\0')){
// endptr==ptr: nichts konvertiert
// *endptr!='\0': hinter Ziffern noch Buchstaben -> falsches Argument
   //Fehler: invalid argument;
   return -1;
}
if(errno){
   die("strtol");
}
//sonst: d weiterverwenden
```

fork & waitpid:

```
pid_t p=fork();
if(p<0){
  die("fork");
}else if(p==0){
  //Kind   -> oder: einzelne Auflistung der arguments:
                    execlp(path,arg1,arg2...,NULL);
  execvp(path,arguments);
  // path: Programm zum Ausführen
  // arguments: const char *arguments[];
  //   -> wichtig:
  //   • arguments[0] = path!
  //   • letzter Pointer in arg: NULL!
  die("execvp");
}else{
  //parent
}
```

```
waitpid:
 int status;
pid_t pid=waitpid(p,&status,flags);
//p=ID von Kind, auf die parent warten will
  oder: p=-1 => warte auf beliebiges Kind
  flags: 0 oder WNOHANG (return, wenn kein Kind sich beendet hat)
 if(pid<0){
  if(errno==ECHILD){
   //Kind, auf das gewartet wird, existiert
   nicht oder ist kein Kind von diesem parent
   return...;
  }else{
   die("waitpid");
  }
 }
if(WIFEXITED(status)){
  int status_message=WEXITSTATUS(status);
  ...
}else{
  //Kind nicht normal terminiert
  (sondern z.B. Seg-fault...)
}
```

getcwd:

//Achtung: nur notwendig wenn Name wirklich gebraucht (opendir, fnmatch & lstat geht auch mit . bei basename(.))

```
char *copy=NULL;
int i=1;
while(1){
   char cwd=realloc(copy,i*1024);
   if(cwd==NULL){
      free(copy);
      die("realloc");
   }
   if(getcwd(cwd,sizeof(cwd))==NULL){
      if(errno==ERANGE){
         i++;
         copy=cwd;
         continue;
      }else{ free(cwd); free(copy);
      die("getcwd");
      }
   }
   copy=cwd;
   free(cwd);
      break;
}
 //copy weiter verwenden
```

compare-Funktion für qsort:

```
static int compare(const void *a,const void *b){
   char *s1=(char*)a;
   char *s2=(char*)b;
   return (strcmp(s1,s2));
}
```

qsort:

```
qsort(array,num_elements,sizeof(...),compare);
// array: Pointer auf Array, das sortiert wird
// sizeof(...): ein Element (z.B. char* bei Wörterliste)
```

pthread_create & pthread_detach:

```
pthread_t pid;
errno=pthread_create(&pid,NULL,start_function,
                     arg_for_start_function);
 if(errno){
   die("pthread_create");
 }
(1) errno=pthread_detach(pid);
                                    } Ressourcen bei Beendigung
 if(errno){                         } automatisch freigegeben
   die("pthread_detach");
 }
(2) if(pthread_join(p,NULL)!=0){    } auf bestimmten thread
      die("pthread_join");          } warten
   }  Wichtig: Ressourcen selbst freigeben
```

fgets & fgetc:

```
char buffer[MAX_LINE_LENGTH+...
while(fgets(buffer,sizeof(buffer),f)!=N...      // f: FILE* oder stdin
   //Behandlung zu langer Zeilen & Rest wegschmeißen:
   if(strlen(buffer)==(size_t)MAX_LINE_LEN...
     if(buffer[MAX_LINE_LENGTH-1]!=...
       if(fprintf(stderr,"Zeile zu lang\n")<0){
          die("fprintf");
       }
       if(fflush(stderr)!=0){
          die("fflush");
       }
       //Rest wegschmeißen:
       while(1){
         int a=fgetc(stdin);      // -> oder f
         if(a=='\n'){
            break;
         }
         if(a==EOF){              // -> oder f
           if(feof(stdin)!=0){
             break;
           }else{
             die("fgetc");
           }
         }
       }
     }
   ... buffer kopieren oder gleich verwenden (je nach Aufgabe)
}
if(ferror(f)){       // -> od. stdin
   die("fgets");
}
```

strtok:

```
char *arg[MAX_LINE_LENGTH/2 +2];
arg[0]=strtok(buffer,"\t\n");
 while(1){     -> int i=1;
   arg[i]=strtok(NULL,"\t\n");
   if(arg[i]==NULL){
      break;
   }
   i++;
 }
```

Makefile:

```
CC=gcc
CFLAGS=-std=c11 -pedantic -D_XOPEN_SOURCE=700 -Wall -Werror
.PHONY: all clean
all: <alle Programme>
clean:
    rm -f <alle Programme>
clash: clash.o
    $(CC) $(CFLAGS) -o clash clash.o
```

Checkliste:

- Funktionsdeklarationen
- globale Variablen & alle Funktionen static
- fflush nach fprintf:

```
if(fflush(stdout)!=0){
        └> od. stderr
    die("fflush");
}
```

- Semaphoren raufzählen vor jedem return!
- fclose vor jedem return:

```
if(fclose(f)!=0){
    die("fclose");
}
```

- closedir vor jedem return (bei Fehler):

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

- semDestroy(s) am Ende der main & vor jedem exit
- Alle Ressourcen mit malloc auch wieder freen!
- '\n' bei fprintf!
- return am Ende der Funktionen
- exit(EXIT_SUCCESS); am Ende v. main

Vorgänge vor Laufzeit v. Programm:

1. Präprozessor
2. Kompilieren: Übersetzung von C-Code in Assembler-code
3. Assemblieren: Erzeugung v. Maschinencode (.o Datei)
   - Symbol- & Relokierungstabelle für Variablen, Funktionen
4. Binden: Erzeugung v. ausführbarer Datei
   statisch: alle Adressen aufgelöst zu Ladezeitpunkt
   dynamisch: Bibliotheken beim Laden relokiert
   => zur Laufzeit durch Plug-ins ergänzen
   └> bildet von logischen Adressen (relativ zu Basis 0) mit Verlagerungskonstante auf reale Adresse ab

UNIX Dateisystem:

- Verzeichnis = Abbildungstabelle: symbolische Namen -> Indexknotennummern
- Hard link = ein Verzeichniseintrag (feste Verknüpfung von Namen zu Indexknotennummer (surjektiv: 2 verschiedene Namen auf gleichen Inode / nur innerhalb 1 Dateisystem / auf jedes Verzeichnis min 2 + Anzahl Unterverzeichnisse / auf jede Datei min 1, sonst gelöscht)
- Indexknotentabelle = Feld von Indexknoten
  -> sammelt Attribute des Gegenstandes (nicht Namen) └> Metadaten
  - Eigentümer (user ID) & Gruppe (group ID)
  - Rechte (lesen, schreiben...) -> auf Verzeichnis => darf Dateien anlegen/löschen; auf Datei => Inhalt anzeigen
  - Zeitstempel (letzter Zugriff / Änderung)
  - Anzahl an hard links, die darauf verweisen (reference counter)
  - Typ (Verzeichnis, Datei...)

-> wenn Verzeichnis: Zeiger auf Verzeichnis (Tabelle)
wenn Datei: zeigt auf Datei
Dateideskriptor = prozesslokale Integerzahl zum Zugriff auf Datei (immer unterschiedlich)

malloc, realloc, calloc, memset:

```
int *numbers = malloc(5*sizeof(*numbers));     od: sizeof(int)
if(numbers == NULL){
    die("malloc");
}
int *new = realloc(numbers, 10*sizeof(*new));
if (new == NULL){
    free(numbers); (bei exit unnötig, nur bei return -1;)
    die("realloc");
}
numbers = new;
```

calloc:

```
int *numbers = calloc(num_elements, sizeof(*numbers));   od sizeof(int)
if(numbers == NULL){
    die("calloc");
}
```

```
int *numbers = malloc(5*sizeof(*numbers));
if(numbers == NULL){
    die("malloc");
}
memset(numbers, value, sizeof(int));   od sizeof(*numbers);
```

-> initialisiert Array mit Wert value

-> Immer: free am Ende!

mmap

-> Vergrößerung von Adressraum

```
void* mmap(NULL, size_t length, PROT_READ|
           PROT_WRITE, MAP_PRIVATE|
           MAP_ANONYMOUS, -1, 0);
```

-> Rückgabewert = Startadresse oder MAP_FAILED bei Fehler

Traps & Interrupts:

- Traps = synchron, vorhersagbar & reproduzierbar (erneute Ausführung v. gleichem Programm mit gleichen Daten & gleichen Prozessor läuft an gleicher Stelle in gleichen Trap)
  -> interne Ursache
  -> Bsp: Division durch 0, Systemaufruf, falsche Speicheradresse
- Interrupts = asynchron, unvorhersagbar, nicht reproduzierbar
  -> externe Ursache
  -> Bsp: Warnsignale d. Hardware (Energiemangel), Beendigung von Ein/Ausgabe Operation

Behandlung v. Interrupts:

└> muss nebeneffektfrei sein!
└> Prozessorzustand bleibt invariant!
1.) CPU macht totale od. partielle Zustandssicherung (Statusregister, Befehlszähler)
2.) BS macht partielle Zustandssicherung
3.) BS behandelt Ausnahme nach Wiederaufnahmemodell
4.) BS stellt Prozessorstatus wieder her
5.) kehrt zu Programm zurück (iret)
-> Hardwareausnahmen auf Ebene 2 behandelt, Softwareausnahmen auf Ebene 3

Betriebsmittel:

- wiederverwendbar (dauerhaft)
  - teilbar / unteilbar
  - angefordert, belegt, benutzt, freigegeben
  - zB: Hardware: Prozessor, Speicher; Software: Text, Daten (Platzhalter)
- konsumierbar (vorübergehend)
  - aufbrauchbar / vermittelbar (unbegrenzt)
  - produziert, empfangen, benutzt, zerstört
  - zB Signale, Zeit, Energie
  - oft: producer, consumer

Adressräume:

1.) Realer Adressraum
= durch Prozessor definierter Wertevorrat an Adressen $[0, 2^{n-1}]$ (n = Bitbreite)
= wirklicher, physikalischer Adressraum
-> nicht jede Adresse gültig
(Befehlssatzebene: 2)

2.) Logischer Adressraum:
= tatsächlich vom Programm verwendete Adressen
totale Abbildung auf reale Adressen: $f: A_l \to A_r$
-> keine Lücken => linear adressierbar
Maschinenprogrammebene (3)

3.) Virtueller Adressraum:
= partielle Abbildung $f: A_v \to A_r$
-> nicht wirklich vorhanden (Abstraktion), wird Programm vorgespiegelt
-> realer Teil: Hauptspeicher / Vordergrundspeicher
-> virtueller Teil in Ablage / Hintergrundspeicher (swap area)

Adressraumschutz:

1.) vor Laufzeit: Einfriedung / -grenzung / Abteilung:
- Programme laufen in realem Adressraum
- Begrenzungsregister legen Ober- / Untergrenze fest
- Adressüberprüfungshardware (CPU/MMU) prüfen zu Laufzeit

2.) zur Laufzeit: Segmentierung:
- Jedes Programm hat eigenen logischen Adressraum
- Adressverschiebungshardware (CPU/MMU) wandeln zur Laufzeit um: real = logisch + Basis
  └> dynamisches Nachladen möglich

Paging:

- BS bildet von virtuellen auf realen Adressraum ab
- wenn Adresse nicht im Hauptspeicher: Trap (Seitenfehler / page fault)
- sucht Adresse in Hintergrundspeicher & lagert ein
- evtl: lagert vorher andere Seite aus (Ersetzungsstrategie)

-> Abbildungseinheit = Seite: (p,o)
- p = Seitennummer (Index in Seitentabelle: evtl. mehrstufig) -> Abbildung auf Seitenrahmen
- niederwertige bits = offset innerhalb Seite

Vorgehen Paging:

1.) Seitengröße in Hex umrechnen ($4096_{10} = 1000_{16}$)
2.) Adresse in Hex als Vielfaches der Seitengröße angeben: $0x2160 \approx 2000_{16}$ => vorderste Ziffer = Index in Tabelle
3.) mit anderer vorgegebener Adresse Versatz im realen Adressraum rausfinden (zB $8192 = 2 \cdot 4096_{10} = 2000_{16}$)
-> wird auf 1000 abgebildet)
=> 0x2160 -> 0x1160
-> Tabelle -> rw (Rechte)
-> 1 für present bit

alternativ: Seitennummer & Versatz ermitteln bei geg. Größe: ganze Adresse in binär hinschreiben, hinterer Anteil = Offset (je nach Seitengröße: $1024 = 2^{10}$ => 10 bits)
-> in Hex umwandeln (vorne zu 0 auffüllen)
-> Rest vorne = Seitennummer: auch in Hex umwandeln & mit ... auffüllen

Gewichtsklassen Prozesse:

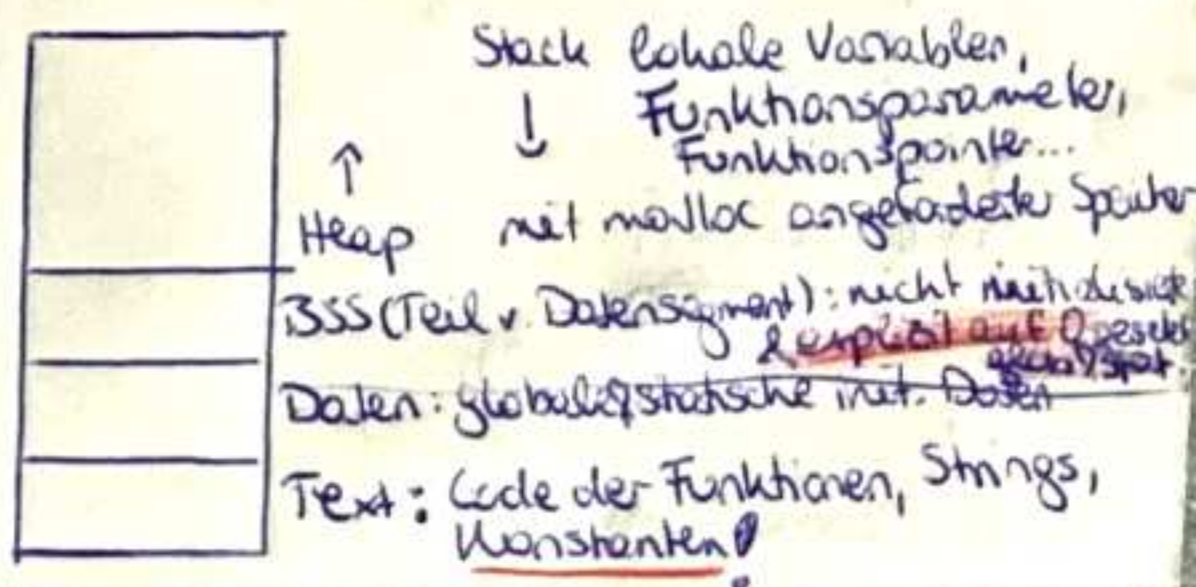
- Federgewichtig:
  - user threads teilen Anwendungsadressraum
    => Umschaltung ohne Adressraumwechsel
  - Vorteil: Erzeugung/Umschaltung billig, Mize...
  - Nachteil: gegenseitige Blockade, keine Nebenläufigkeit
- Leichtgewichtig: -> müssen vom BS speziell unterstützt
  - Prozesse teilen Anwendungsadressraum, aber verschiedene Kern-Adressräume
    => Umschaltung mit 1 Adressraumwechsel
- Schwergewichtig:
  - verschiedene Anwendungs- & Kernadressräume
    => Umschaltung mit 2 Adressraumwechseln

Semaphor:

= Spezieller Datentyp (Synchronisationsvariable) zur Koordination von Threads oder zur Organisation von Ressourcen
-> 2 intrinsische Eigenschaften:
- P verringert um 1 (down / wait / acquire), blockiert bei (1->0)
  -> von laufend zu blockiert
- V erhöht (up / signal / release) um 1
  -> blockiert -> bereit (0->1)

Zustände von Prozessen:

Aufbau logischer Adressraum:



Speicherzuteilung:

- Laufzeitsystem: lokale Verwaltung von Speicher, der Prozess zugeteilt wurde
  -> feingranular, sprachorientiert
- Betriebssystemebene: Verwaltung von globalen Haupt- & Arbeitsspeicher für alle Prozesse
  -> grobgranular, systemorientiert

Freispeicherlisten:

- best fit: Größe ↑
  (+) min. Verschnitt (kleinstes Loch, das passt)
  (−) lange Suche, Verschmelzung & Einsortierung aufwändig
- worst fit: Größe ↓
  (+) min. Suchaufwand
  (−) großer Verschnitt, Einsortierung & Verschmelzung aufwändig
- first fit: aufsteigende Adressen
  (+) min. Aufwand - laufzeiteffizient, Verschmelz. & Einsort. einfach
  (−) viel Verschnitt
- next fit: aufsteigende Adressen, Start ab letzter Fundstelle
  (+) wie first fit & weniger Verschnitt

Programm, Prozess:

- Programm = für Maschine konkretisierte Form v. Algorithmus
- Prozess = Programm in Ausführung mit Daten
- Prozessinstanz (-inkarnation) = physische, konkrete ...