File öffnen + auslesen

```c
FILE *file = fopen(path, "r");
if (!file) die("fopen");
char buf [LENGTH+1];
while(fgets(buf, sizeof(buf, file/stream)){
  ...}
if(ferror(file)) die ("fgets");
if(fclose(file)==-1) die("fclose");
```

einlesen stdin:

```c
char ** wlist = NULL;
char word-buf [103];
int count = 0;
while(fgets(word_buf, sizeof(word_buf), stream/file)){
if(strlen(word-buf)>100){
  int c;
  while(c=fgetc(stdin)){
      if(c ==(int) '\n' || c== (int) EOF)
        {break;}
  }
if(feof(stdin){break;} die("fgetc");}}
  //Aufruf {
if(ferror(stdin)) die("fgets");
fclose(file);
```

strtok:

```c
char* args [maxlength/2];
char* cpy = strdup (input);
  if(!cpy) die("strdup");
for (int i = 0; i< (maxlength/2); i++){
  if (i==0){args[i] = strtok(input, "_\t");}
  else{args[i] = strtok[NULL, "_\t");}
  if(!args[i]) {break;}}
```

strtol:

```c
errno = 0
char * endptr;
long x = strtol(str, &endptr, 0);
if(errno) die ("invalid number");
if(*endptr != '\0' || x<=0 || x >INT_MAX){
  fail("invalid number");}
int y = (int) x;
```

fail/die

```c
static void fail(const char* msg){
  fprintf(stderr, "msg");  ← fail
  perror("msg");  ← die
  exit(1);
```

basename/dirname

```c
char*cpy = strdup(path);
  if(!cpy) die("strdup");
char * base = basename(path); ->filename
              dirname(path); ->dir name
qsort(input, amount, sizeof(char*), cmp);
static int cmp(const void*a, const void*b){
  return *((const int*)i1) - *((const int*)i2)};
  ↑ aufsteigend -> für absteigend Argumente tauschen ①
//andere Mglk:
  return strcmp((const int*)i1, (const int*)i2);}
if(fflush(stdout)) die ("fflush");
```

fnmatch

```c
int fn =                      ==EOF
fnmatch(pattern, text, NULL/FNM-PERIOD/FNM-PATHNAME);   ==0   "/" ↳ gekoppt
if(rv && FNM_NOMATCH){fail("fnmatch");
```

main + threads

```c
int main(int argc, char* argv[]){
  //argc prüfen ->usage
sem = semCreate(1);
  if(!sem) {die("semCreate")};
  P(sem );
  pthread_t pid;
errno = pthread_create(&pid, NULL, start, arg)){
  if(errno) die ("pthread_create");
errno = pthread_detach(pid);
  if(errno) die (" pthread_detach");
//ohne detach ~> passiv warten
pthreads_t pids [MAX_THREADS];
  -> pthread_create
for (int i=0; i< MAX_THREADS; i++){
  errno = pthread_join(tids[i], NULL);
  if(errno) die ("pthread_join");
//aktiv warten
  while(1){ P(block)
  if (counter ==0) break;
  V(block);
//passiv warten
  while (count >0){ => extra sem(wait) mit 0 initialisieren
    P(wait);
    count --;}
  oder
  for (int i=0; i< count; i++){
    P(wait);}
//free();
// semDestroy();
//fflush();
  exit(0);
```

in start ggf. casten
char *a = (char*) arg;

```c
char* buf = malloc(size);
  if(!buf) die ("malloc");
char*ptr = getcwd(buf, size);
  if(!ptr) die("getcwd");
  if(errno == ERANGE){
    buf = realloc(buf, size + x);
    if(!buf) die("realloc");
get_current_dir_name();
```

Eingabe reinigen

```c
while(1){
  int tmp = fgetc(stdin);
  if(tmp == '\n') break;
  if(tmp == EOF){
    if(feof(stdin)){
    // Behandlung
  }else{
    //errno Behandlung
  }}}
```

- '\n' am Ende bei printen!
- exit(0);
- returns
- free/semDestroy, fflush
- Static!!!

öffnen Verzeichnis / **durchlaufen**

```c
DIR * dir = opendir(path);
  if(!dir) die("opendir");
struct dirent* entry;
while(errno=0, (entry=readdir(dir))){
  char* name = entry->d_name;
  if(strcmp(".", name)==0 || strcmp("..", name)==0){
      //Auftrag meist continue;
  char lpath[strlen(path)+ strlen(name)+2];  //bei Threads strdup
  if(snprintf(lpath, strlen(lpath), "%s/%s", path, name)<0){
    fail("snprintf");
  struct stat buf;
  if (lstat(path, &buf) == -1){ die("lstat"); 
    if(S_ISREG(buf.st_mode){
      //Datei
    }
    if(S_ISDIR(buf.st_mode){
      //Verzeichnis durchlaufen ~> Rekursion
    },
  }
if(errno){die("readdir");}
if(closedir(dir)==-1){die("closedir");}
```

LSTAT

Kindprozesse erzeugen

```c
pid_t fork();
switch (pid){
case -1:
   die("fork");
   break;
case 0: //Kind
   execvp(cmd[0], cmd);
   die("execvp");
   break;
case default: //Parent
   int wstatus;
   waitpid(pid, &wstatus, 0); / if(wait(&status)==-1){ die("wait");
   break;
   if (WIFEXITED (wstatus)){
   int code = WEXITSTATUS(wstatus);
```

① char*const* a1 = (char*const*)a;
return strcmp(*a1, *b1);

Zombies

```c
while (1){
  int status;
  pid_t p = waitpid(-1, &status, WNOHANG);
  if (p==0) break;
  if (p==-1){if (errno == ECHILD) break;
    die ("waitpid");}
  if (WIFEXITED (status)){
    int code = WEXITSTATUS(status);}
```

Makefile

```make
CC=gcc
CFLAGS = -std=c11 -pedantic -Wall
         -Werror -D_GNU_SOURCE -g
                      ^ ggf. -pthread
.PHONY all clean
all: creeper
clean: rm -f creeper creeper.o
creeper: creeper.o argparser.o
   $(CC) $(CFLAGS) -o creeper creeper.o argparser.o
creeper.o: creeper.c argparser.h
   $(CC) $(CFLAGS) -c creeper.c
```

- lok. Variablen
  - ↳ static: Speicherplatz für gesamte Programmausführungszeit reserviert
  - ↳ auto: Speicherplatz bei Betreten eines Blocks reserviert, danach freigegeben ~> uninit = undef. Wert

- Stat. Binden: fehlende Fkt. aus Bibl. genommen + einkopiert
  - zum Ladezeitpunkt alle Referenzen aufgelöst → sehr groß
  - → Komp. & Ass.: lokale Ref.
  - → Binder: globale Ref.
- dyn. Binden: Fkt. aus gemeinsam nutzbaren Fkt. wird nicht einkopiert => erst bei Bedarf eingebunden
- Maschinenbefehle: -> direkt interpret. durch CPU
  - ↳ part. interpret. durch BS
- Systemaufrufe: -> part. interpret. durch BS

- Adresslänge: 16 Bit
- Seitengröße: 4096 = 2^{12} Byte
- Seite an virt. Adresse 8192 auf Kachel an Adresse 4096 (phy)

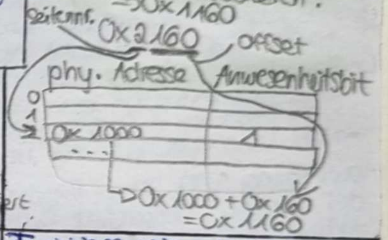
  => Abb. 0x 2160 auf phys. Adr.?

  2. Eintrag in Seitentabelle ~> auf 4096 abgebildet

  offset wg. 12

  => 4096 ≙ 0x 1000

  ↳ offset aufaddieren: => 0x 1160



Funktionsweise Anwendungsprog. mit Speicherzuteilung:

-> Anwendungsprog. (Maschinenprog.-Ebene) fordert Speicher v. LZS durch Aufruf malloc(); wenn nicht mehr gebraucht -> über free() an LZS zurückgegeben
=> hat LZS-Speicherverwaltung nicht ausreichend Speicher => Anforderung größere Blöcke vom BS mit mmap/brk() => damit malloc()-Anfragen bedient werden

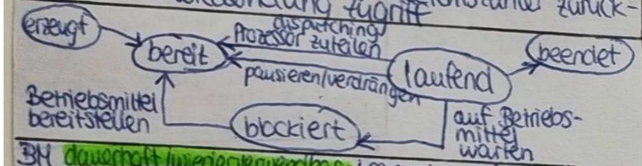
- privilegierte Maschinenprog. -> durch BS implementiert => Benutzerebene
- unpriv. Op. direkt ausgeführt
- priv. Op. ~> Moduswechsel/Systemaufruf
- Systemebene: /Ausführungsplattform
  - alle Maschinenbefehle direkt ausgeführt

Trap: interne Ursache, vorhersagbar, reproduzierbar,
Abfangung synchrone Unterbrechung: (unbekannter Befehl, Adressraumverletzung, falsche Rechenoperation Systemaufruf) Speicherzugriffe

Interrupt: externe Ursache, unvorhersagbar, nicht reprod.
Unterbrechung asynchrone Unterbrechung: (Signalisierung „externer" Ereignisse (Taste), Beendigung E-/A-Operation
↳ **Wiederaufnahmemodell**: Ausführungsfortsetzung nach erfolgter & erfolgreicher Behandlung: page fault
=> bei Trap optional, bei Interrupt zwingend
↳ **Beendigungsmodell**: Ausführungsabbruch
=> Ausnahmesituation kann nicht behandelt werden
=> Abbruch d. Programms (Zugriffsverletzung)
=> bei Trap optional, bei Trap <u>nicht erlaubt</u>

Abwicklung CPU sichert Register -> BS ruft Händler auf der Situation behandelt <-> Programm wird beendet,
=> Wiederherstellung Befehlszähler & Register durch CPU
=> nimmt Ausführung wieder auf

Pagefault: Zugriffsfehler -> Trap -> MMU sendet pagefault
-> Seite einlagern (durch BS) -> Befehlszähler zurückgesetzt -> Wiederholung Zugriff



BM **dauerhaft/wiederverwendbar**: begrenzt angefordert, teilbar/unteilbar => angefordert/belegt/benutzt/wieder freigegeben
zeitlich
-> Unteilbare BM = multilaterale Synchronisation
Bsp.: Prozessor (CPU, MMU), Speicher (RAM), Periphere Ein- & Ausgabe, Text, Daten

konsumierbar/unbegrenzt: unbegrenzt
-> produziert/zugeteilt/benutzt/aufgebraucht
Bsp. Signal (trap), Datenstrom, Zeit, Energie
↳ konsumierbare BM = unilaterale Synchronisation

- **Synchronisation** erforderlich bei Prozessen mit gleichzeitigem Zugriff auf teilbare & konsumierbare BM
- **Semaphor**: Koordination v. Threads, Datenstruktur mit (atomarer) Zählvariablen => Koordination in einem mehrfädigen Programm
  ↳ P( ): dekrementiert Zähler, blockiert Aufrufer, falls Zähler ≤ 0
  ↳ V( ): inkrementiert Zähler, weckt ggf. wartende threads
  (↳ semCreate( ); semDestroy)
  kritischer Abschnitt: es darf sich immer nur ein Prozess hier befinden

schwergewichtig: Prozessinkarnation & Benutzeradressraum bilden eine Einheit -> Prozesswechsel: 2 Adressraumwechsel

leichtgewichtig/Kernel-Level: Entkopplung v. Prozessinkarnation & Adressraum => 1 Adressraumwechsel • effiziente Steuerumschaltung
-> Gruppe v. Threads nutzt gemeinsame BM • Ausnutzung Multiprozessor
-> jeder Thread ist BS bekannt

federgewichtig/User-Level: Prozessinkarnation & Adressraum bilden eine Einheit -> kein Wechsel • billig
-> Systemkern hat kein Wissen => gegenseitige Blockade => Umschaltung in Systemkern
-> eines geblockt => alle geblockt

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

Bibliotheken/Binden:
Präprozessor -> Compiler -> Assembler -> Linker
statisch: einkopiert zu Übersetzungszeitpunkt
dynamisch: nicht einkopiert zum Ladezeitpunkt, Plugins möglich
Seitendeskriptor -> Zugriffsrechte, Bit, ob Seite ausgelagert
Filedeskriptor -> prozesslokale Integerzahl zum Zugriff auf eine Datei
- **Segmente**: verschieden groß (Attribute: Granularität, Adresse, Länge) ⇝ Länge ∧ Offset (2D)
- **Kacheln**: paging: gleich groß, unterschiedliche Adressen ⇝ eindimensionales Adressraum
  -> Adressraum in Seiten aufgeteilt -> auf Seitenrahmen im
- **Paging**: Basisregister (BS) zeigt auf (aktuelle) Seitentabelle (MMU) mit Platz für 2^n Einträge
  ⇝ zeigen auf Seitenrahmen (Adresse = Seitendeskriptor + Offset)
  ⇝ realer Adressraum = Aufteilung in Kacheln

Schutz durch log. Adressräume:
-> Schutz d. BS vor Eingriffen/Schutz anderer Prozesse vor willkürlichem/versehentlichem Überschreiben/
Schutz d. eigenen Prozesses vor z.B. Adressierungsfehlern
Schutz durch Segmentierung ⇝ dyn. Speicherverwaltung mgl.
⇝ jeder Prozess hat eigenen Adressraum (log.)
Schutz durch Abgrenzung: ⇝ unpriv. Modus: Gatter aktiv ⇝ priv. Modus: Änderungen am Inhalt erlaubt
-> Schutzgatter trennt den vom residenten Steuerprogramm vom Programm belegten Hauptspeicher

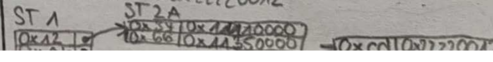
Speicherzuteilung
Laufzeitsystem verwaltet den dem Prozess zugeteilten Speicher lokal & eigenständig ⇝ sprachenorientierte Kriterien ⇝ feingranulare Zuteilung
Betriebssystem verwaltet gesamten Hauptspeicher d. gesamten Systems ⇝ global für alle Prozesse
⇝ systemorientierte Zuteilung ⇝ grobgranular
Zusammenarbeit: Anwendungsprogramm (Maschinenprog.-Ebene) fordert Speicher v. LZS durch Aufruf v. malloc an, wenn nicht mehr gebraucht ⇝ über free() zurück an LZS ⇝ hat LZS nicht ausreichend Speicher ⇝ Anforderung von größeren Speicherblöcken vom BS (brk() / mmap())
⇝ malloc Anfragen bedient werden
=> Anwendungsprogramme, Laufzeitsystem (LZS), BS, Maschinenprogramm = AP + LZS
- **sequentiell**: zeitlich nicht überlappend/deterministisch
- **parallel**: nicht sequentiell => Wettlaufsituation

Betriebsarten
Mehrzugang: CPU & Speicherschutz wichtig
Mehrprogramm: mehrere Programme
Stapel ⇝ top to bottom
Echtzeitbetrieb: • vorhersagbares Laufzeitverhalten
• Einhalten v. vorgeg. Terminen durch physik. Weg
• Rechtzeitigkeit vor Geschwindigkeit
• Überwachung & Interaktion v./mit externen Prozessen
Terminvorgaben
weich: Ergebnis weiterhin von Nutzen, verliert mit weiterem Zeitverzug zunehmend an Wert, Terminverletzung tolerierbar
fest: Ergebnis wertlos => verworfen, interne Prozess abgebrochen und erneut bereitgestellt, tolerierbar
hart: Verspätung d. Ergebnislieferung kann zu Katastrophe führen => internem Prozess wird Ausnahmesituation zugestellt
Terminverletzung <u>nicht</u> tolerierbar, aber möglich

call-by-value: Kopie d. tatsächlichen Parameters wird an Funktion übergeben
Speichervirtualisierung: Mehrprog. betrieb (BS)
-> **Ladestrategie**: wann muss Datei im Hauptspeicher liegen?
-> **Freisetzungsstrategie**: welches Datum im HS ersetzbar?
- **best-fit**: Verschnitt minimieren ⇝ aufsteigende Größe ⇝ Lochmenge sortiert
- **worst-fit**: Suchaufwand ↓
- **first-fit**: Laufaufwand ↓ ⇝ aufsteigende Sortierung nach Speicheradresse
- **next-fit**: Verschnitt nivellieren
  => Verschmelzung

Einlastung = Moment, indem CPU umgeschaltet wird => Prozesswechsel
Begrenzungsregister legen Unter-/Obergrenze eines Programms im Adressraum fest (CPU)
real: nicht alle Adressen gültig und zur Programmspeicherung verwendbar, wird durch Hardware vorgegeben
logisch: (durch BS) vollständig auf Hauptspeicher abgebildet; Adressraum, indem sich einzelner Prozess bewegt, zusammenhängend, linear adressierbar, nur gültige Adressen => keine Lücken
virtuell: nicht jede Seite muss präsent im (MMU) Speicher sein, da diese ausgelagert werden können => swapping area, wird auf Hauptspeicher abgebildet, allerdings nicht vollständig, nur gültige Adressen

Klausurfragen – Theorieteil:
- Adressraumerweiterung Laufzeit, Schnittstelle + Nutzung nur im Spezialfall, eigentl. Schnittstelle
  -> größere Datenmengen zur Laufzeit benötigt -> Benutzereingaben
  -> mmap(): Systemaufruf grobgranular & langsamer arbeiten als malloc()
  -> dynam. Speicherallokation = gute Mglk. zur flexiblen Speicherverwaltung
  -> malloc()/realloc()
- inodes: ⇝ eindeutige num. Adresse im Namensraum
  -> bei File ?
  -> bei Symlink: in Symlink selbst => <u>hinterer</u> Teil
  ↳ in Verzeichnis: vorderer Teil
- Pagefault:
  1) trap -> page fault (MMU) -> Zugriffsfehler + synchr. Programmunterbrechung
  2) Seite einlagern 3) Zugriff wiederholen
- Inodes: Eigentümer (UID), Gruppenzugehörigkeit (GID), Rechte, Zeitstempel, Anzahl d. Verweise (hard-links)
- Paging/Seitentabelle:
  1) log. Adresse aufteilen in 2er Paar (von links)
  2) entsprechende Indices in Seitentab. wählen
  3) Page-offset aufaddieren
- SegFault: Zugriff auf nicht initialisierten/ungültigen Speicher
- Virt. AR -> log. AR:
  - Befehlssatzebene fängt Zugriff auf ausgelagerte Seite/Segment ab => Weiterleitung an BS => trap + ggf. Page Fault
- readdir & Stat:
  -> Probleme mit Symlinks: wenn stat(), dann Rückgabe d. Dateiinfos d. verbundenen Datei => bei toten Symlinks zufällig
  -> inodes einzigartig innerhalb eines Dateisystems
  -> bei übergreifendem Zugriff kann es zu Problemen kommen
  -> unregelmäßiger Zugriff, randomisiert, mgl. SegFault
- zweiseitige Seitentabelle:
  0x1234 0000 -> 0x1111 0000
  0x1266 0000 -> 0x1135 0000
  0xabcd 0012 -> 0x2222 0012



Stack: lokale <u>nicht static</u> Var + Funktionsparam
Heap: dyn. Erweiterung BSS Segment
BSS: <u>uninitialisierte</u> lok & glob. static Variablen (=0)
(init) Daten: <u>initialisierte</u> lok & glob. static Var.
Text: Instruktionen d. main, konstanten

- SegF => Prozess beendet, SIGSEGV zustellen
- Adressraumschutz durch Eingrenzung
  -> Begrenzungsregister legen Adressbereich im phy. AR fest, auf den die Speicherzugriffe beschränkt
- UNIX-Systemaufruf,
  mithilfe v. Systemaufrufen kann Benutzerprog. priv. Operationen durch das BS ausführen lassen, die es im normalen Ablauf nicht selbst ausführen darf
- User-/Kernel-Level:
  -> ... können anwendungsspez. Scheduling strategisch eingesetzt werden
- aktuelle Arbeitsverzeichnis
  -> jedem UNIX-Prozess
  -> Pfadnamen, die mit / beginnen, werden relativ zum aktuellen Verzeichnis interpretiert
- Stat./dyn. Binden:
  -> dyn. Binden: Fehlerkorrekt. in Bibl. leichter übernommen werden, da nur Bibl. neu erzeugt, Prog. die Bib. verwenden, müssen <u>nicht</u> neu komp. & gebunden werden
- Prozessoren unpriv. Benutzermodus/priv. Systemmodus
  -> im Benutzermodus ist Menge an ausführbaren Prozessorbefehlen eingeschränkt
- Hacker-Angriff:
  -> strcpy(), strcat() ↳ spezielle Befehle zur Ansteuerung externer Geräte stehen nur im privilegierten Modus zur Verfügung
- Seitendeskriptor: einer Seite d. virt. AR
  -> Zugriffsrechte
- interrupt:
  -> Signalleitung: Prozess Unterbrechung angezeigt
  -> Prozessor sichert aktuellen Zustand bestimmter Register inkl. Programmzähler, springt in vordefinierte Behandlungsfkt.
- Systemaufruf:
  -> Parameter werden nach fester Konvention an das System übergeben
- Programmbibliothek:
  -> Programm-Module, die v. mehreren Anwendungen gemeinsam genutzt werden, können in Form einer dyn. Bibliothek zentral installiert werden, um Speicherplatz zu sparen
- Seitennummer + Versatz: Seitengröße
  -> 1024 = 2^{10} ⇝ 0xcafe => 16 Stellen -> 10 Stellen
  1100 1010 | 1111 1110
  3 2 2 f e
  Seitennr. Versatz
- dyn. Binden:
  -> dyn. gebundene Programme können noch zur Laufzeit durch das Nachladen neuer Programmmodule/Plug-ins ergänzt werden
- Adressraumverwaltung:
  -> Speicherbereich, der mit free() freigegeben wurde, verbleibt im log. Adressraum d. zugehörigen Prozesses
- Speicherverwaltung:
  -> Verwendung gekachelter Speicherverwaltung, alle Kacheln gleich groß
- Adressraumschutz:
  -> Adressraumschutz durch Segmentierung mgl., dass dieselbe log. Adresse in unterschiedlichen log. Adressräumen auf untersch. phys. Adr. verweist
- Adressabb. virt. AR
  -> Prog. adresse -> log. Ad. -> virt. Ad. -> reale A.