Systemprogrammierung

2022/23

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) ;
```

Rechnerorganisation

Übersetzung:

1. Schritt: Präprozessor
– Kommentare werden entfernt
– Bearbeitung von Include und Macros
2. Schritt: Compilieren
– C wird in Assembler übersetzt
3. Schritt: Assemblieren
– Maschienencode wird erstellt
4. Schritt: Binden
– Ausführbare Datei wird erstellt
– statisch und dynamisch möglich (betrifft Bib-Funktionen)

Grundbegriffe:

– Programm: Folge von Anweisungen
– Prozess: In Ausführung befindliches Programm
  • Kann mehrere Programme ausführen
– Kompilierer: Wandelt Quellsprache in Zielsprache um
– Interpreter: Führt Programm direkt aus

Laden eines Programms:

– Statistisch gebundene Programme:
  • Von Ebene 5 auf Ebene 3
  • Läd beim Compilieren alle Komponenten in den Speicher
  • Alle Adressen werden zum Bindezeitpunkt aufgelöst
– Dynamisch gebundene Programme:
  • Von Ebene 5 auf Ebene 1
  • Läd zur Laufzeit benötigte Komponenten
  • Adressen werden beim Programmstart aufgelöst
  • Auflösung von Adressbezüge beim Übersetzen möglich

Mehrebenenmaschine:

– 5. Problemorientierte Programmierebene
– 4. Assembliersprache
– 3. Maschinenprogrammebene
– 2. Befehlssatzebene
– 1. Mikroarchitekturebene
– 0. digitale Logikebene
– Hinweise:
  • Schichten 3-5 repräsentieren virtuelle Maschine
  • Schichten 0-2 repräsentieren reale Maschine
  • Schicht 4 Logisch existent aber unwichtig geworden
– Durchlauf:
  • $5 - 4$: Kompilation
  • $4 - 3$: Assemblieren und Bindung
  • $3 - 2$: Partielle Interpretation
  • $2 - 1$: Interpretation

Programmhierachie:

– Maschinenprogramme enthalten zwei Befehlsformen:
  • Maschinenbefehle der Befehlssatzebene
  • Systemaufrufe an das Betriebssystem
– Maschinenprogramme (MaschProg):
  • ohne Compiler vom Prozessor ausführbar
  • Meist von Compiler generiert
  • Grundlage bilden Hochsprachen und Assembler
– Triumvirat:
  • MaschProg: Anwendungsprogramm + Laufzeitsystem
  • MaschProg: Benutzerebene
  • Betriebssystem
  • Zentraleinheit
  • Ausführplattform: Betriebssystem + Zentraleinheit
  • Ausführplattform: Systemebene

Betriebssystem

– Menge von Programmen der Befehlssatzebene
– Stellt ein nicht sequentielles Programm dar
– Zählt nicht zur Klasse der Maschinenprogramme
– interpretiert die eigenen Programme nur teils partiell
– sollten unterbrechbar sein
– Interpreter - Ausführung von Systemaufrufen:
  • 1. Prozessorstatus unterbrochener Programme sichern
  • 2. Systemaufruf interpretieren
  • Prozessorstatus wiederherstellen

Maschinenprogrammebene:

Zwei Sorten von Befehlen:
– unpriviligierte Befehle:
  • Wird von der CPU direkt ausgeführt
– priviligierte Befehle:
  • Werden vom Betriebssystem ausgeführt
  • explizit als Systemaufrufe codiert
  • implizit als Ausnahmen ausgelöst

Ausführungsablauf bei priviligierten Befehlen:
  • 1. CPU interpretiert Maschinencode befehlsweise
  • 2. Bei Ausnahmen startet das Betriebssystem
  • 3. Interpretiert Programme des BS befehlsweise
  • Folge von Punkt 3:
  • 4. BS interpretiert Maschinencode befehlsweise
  • 5. CPU wird zum weitermachen verleitet

Ausnahmebehandlung:

– Ausnahmebehandlung ist zwingend
– Es gibt zwei Varianten:
– 1. Trap
  • Abfangung für Ausnahmen von interner Ursache
  • Z.b. Bug im Code
  • falsche Adressierungsart oder Rechenoperation
  • Systemaufruf, Adressraumverletzung,unbekanntes Gerät
  • Seitenfehler im Falle lokaler Ersetzungsstrategien
  • synchron, vorhersagbar, reproduzierbar
  • Behandlungsmodelle: Beendigung und Wiederaufname
– 2. Interrupt:
  • Unterbrechung durch Ausnahmen von externer Ursache
  • Ein externer Prozess signalisiert einen Interrupt
  • Z.b. Beendigung einer DMA- bzw. E/A-Operation
  • Z.b. Seitenfehler im Falle globaler Ersetzungsstrategien
  • Unterbrechungsbehandlung muss Nebeneffektfrei sein
  • Aktiver Prozess wird unterbrochen, nicht abgebrochen
  • asynchron, unvorhersagbar, nicht reproduzierbar
  • Behandlungsmodell: Nur Wiederaufnahmemodell

Aktives Warten:
– Prozess frägt beim warten immer wieder nach
– Vergeudet nicht unbedingt CPU-Zeit
– Benötigt keine Unterstützung durch das Betriebssystem

Passives Warten:
– Prozess Schläft, bis er benachrichtigt wird

Betriebsarten

Stapelbetrieb:

Abgesetzter Betrieb:
– Verwendung von Satellitenrechner und Hauptrechner
– Satellitenrechner sendet Eingabe an Hauptrechner
– Hauptrechner verarbeitet die Eingabe
– Hauptrechner schickt Ergebnis an Satellitenrechner
– +: Entlastung durch Spezialrechner

Überlappte Ein-Ausgabe:
– Durch Interrupts gleichzeitige Prozesse
– Speicherdirektzugriff (DMA) statt programmierte IO
– nebenläufige Programmausführung möglich
– Problem: Leerlauf beim Auftragswechsel

Überlappte Auftragsverarbeitung:
– Verarbeitungsstrom auszuführender Programme
– Technik: Prefetching oder Auftragseinplanung
– Probleme: CPU Monopolisierung, IO Leerlauf

Abgesetzte Ein-Ausgabe: Spooling
– Abwechselnd: CPU-Stoß und IO-Stoß
– Entkopplung durch Pufferbereiche

Mehrprogrammbetrieb:
– Multiplexen der CPU
– Mehrere Programme gleichzeitig im Hauptspeicher
– Nutzung von Aktiven/Passiven Warten
– Wichtiges Tool: Adressraumschutz:
  • Vor Laufzeit: Schutz durch Eingrenzung
  • Zur Laufzeit: Schutz durch Segmentierung

Dynamisches Laden:
– Problem: Programm zu groß für den Arbeitsspeicher
– Lösung:
  • Zerteilung des Programms in kleine Teile
  • Das Nachladen ist Programmiert
  • Nachteil: Programmierer manuell einzubauen

Echtzeitbetrieb:

– System muss ständig betriebsbereit sein
– Ergebnisrückgabe innerhalb vorgegebener Zeit
– externe (physikalische) Prozesse definieren Verhalten für nicht termingerechte Ergebnisrückgabe

Terminvorgabe:
– weich:
  • Ergebnis weiterhin nutzbar
  • Ergebnis wird nur mit der Zeit immer wertloser
  • Terminverletzung ist tolerierbar
– fest:
  • Ergebnis ist wertlos
  • Ergebnis wird verworfen
  • Terminverletzung ist tolerierbar
– hart:
  • kein Ergebnis bedeutet großes Problem
  • Terminverletzung ist nicht tolerierbar

Mehrzugangsbetrieb

– Nur sinnvoll mit CPU- und Speicherschutz

Dialogbetrieb:
– mehrere Benutzer gleichzeitig
– Benutzereingaben und Verarbeitung wechseln sich ab
– Zugang über Dialogstation (z.b. Terminal)
– Problem: Monopolisierung der CPU möglich

Dialogorientiertes Monitorsystem (Hintergrundbetrieb):
– Prozesse im Vordergrund starten
– Prozesse im Hintergrund vollziehen
– mehrere Aufgaben werden parallel bearbeitet
– Problem: Hauptspeichergröße

Teilnehmerbetrieb:
– eigene Dialogprozesse werden interaktiv gestartet
– Zeitscheibe um CPU-Monopolisierung vorzubeugen

Teilhaberbetrieb:
– Client-Server-System

Systemmerkmale

Symmetrische Simultanverarbeitung:
– Mehrere Prozesse gekoppelt über gemeinsames Verindung
– Stellt homogenes System dar

Speichergekoppelter Multiprozessor:
– Alle Prozesse nutzen den Hauptspeicher
– Stellt heterogenes System dar
– Assymetrische Architektur:
  • hardware bedingter assymetrischer Betrieb
– Symmetrishce Architektur:
  • Betriebssystem legt Multiprozessorbetrieb fest

Parallelverarbeitung:
– N Prozessoren können:
  • N Programme echt parrallel ausführen
– Jeder dieser Prozessoren kann multiplexen
  • Jeder Prozessor kann einzeln parallelisiert werden

Schutzvorkehrungen:
– Jeden Prozessadressraum in Isolation betreiben
– Prozessen eine Zugriffsbefähigung erteilen
– Objekten eine Zugriffkontrolliste geben
– 1. Methode: Schutz durch selektive Autorisierung
– 2. Methode: Zugriffsrechtmatrix
– 3. Methode: Schutzring names Multics
  • Einzelne Ringabschnitte mit unterschiedlichen Rechten
  • Von außen nach innen mehr Rechte
  • ring fault bei Rechteverletzung
– 4. Methode: Schutzgatterregister:
  • Permanent benötigte Programme werden isoliert
  • Abgrenzung von den Anwendungsprogrammen
  • Schutzgatter kann manuell ausgeschalten werden

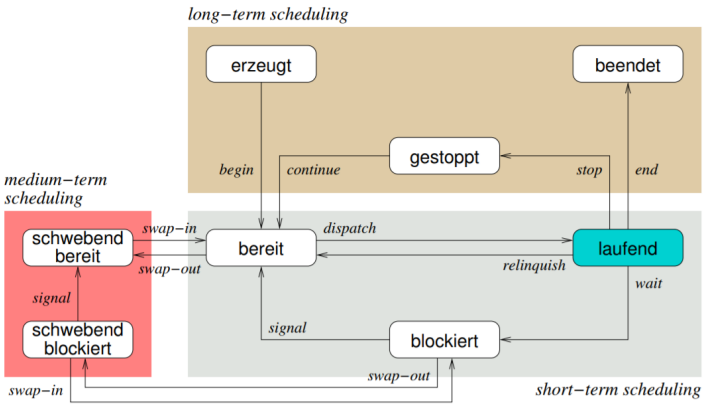
Umlagerung nicht ausführbereiter Programme:
– auch swapping genannt
– schafft Platz im Arbeitsspeicher
– Probleme: Fragmentierung, Verdichtung

Umlagerung ausführbereiter Programme:
– Nicht benötigte Teile werden ausgelagert
– Zugriff auf ausgelagerte Teile unterbricht Prozess
– Intensiver Wechsel zw. aus und einlagern

Prozessverwaltung

Begriffe:
– Einplanung: Reihenfolenbildung von Aufträgen
– Einlastung: Zuteilung der Betriebsmittel

Programmfaden:
– Einplanungseinheit für die Prozessorvergabe ist der Faden
– Lauf- und Wartephasen betreiben einen Rechner stoßartig
– Fäden überdecken passives Warten anderer Fäden



Kurzfristige Planung (short-term scheduling):
– Wichtig für Mehrprozessbetrieb
– bereit: Prozess ist in der Bereitliste
– laufend: Prozess vollzieht seinen CPU-Stoß
– blockiert: Prozess erwartet Betriebsmittelzuteilung

Mittelfristige Planung (medium-term scheduling):
– Anhand der Umlagerung kompletter Programme
– schwebend bereit:
  • Das Exemplar eines Prozesses ist augelagert
  • Die Einlastung des Prozesses ist außer Kraft
– schwebend blockiert:
  • ausgelagerter ereigniserwartender Prozess

Langfristige Planung (long-term scheduling):
– Nutzung von Lastkontrolle
– erzeugt: fertig zur Programmverarbeitung
– gestoppt: erwartet Fortsetzung / Beendigung
– beendet: Prozess erwartet seine Entsorgung

Gütemerkmale:
– Benutzer:
  • Antwort-/Durchlaufzeit, Termine, Vorhersagbarkeit
  • Prozessausführung unabhängig von der Systemlast
– System:
  • Durchsatz, Auslastung, Gerechtigkeit
  • Dringlichkeit, Lastausgleich
  • Gleichmäßige Betriebsauslastung

User-Threads:
– Threads mit nichtpriviligiertem Code
– blockierende syscalls blockieren andere User-Threads
– Schedulingstrategie vom Programmierer definiert
– Können effizient umgeschalten werden
– Nicht für Multiprozessorbetrieb ausgelegt

Kernel-Threads:
– Threads mit priviligiertem Code
– Erzeugung teurer als User-Threads
– blockierende syscalls blockieren keine anderen Threads
– Schedulingstrategie durch Betriebssystem vorgeben
– Umschalten findet immer im Systemkern statt
– Kann für Multiprozessorbetrieb sinnvoll sein

Klassifikation der Prozesseinplanung:
– kooperative Planung (FCFS):
  • für voneinander abhängige Prozesse
  • CPU-Entzug nicht zugunsten anderer Prozesse
  • laufender Prozess gibt CPU nur mit Systemaufruf ab
  • CPU Monopolisierung möglich
– preemptive Planung (RR, VRR, SRTF):
  • für voneinander unabhängige Prozesse
  • CPU-Entzug zugunsten anderer Prozesse möglich
  • ereignisbedingte Verdrängung laufender Prozesse
  • CPU Monopolisierung nicht möglich

– deterministische Planung:
  • Alle Prozesszeiten sind bekannt
  • Einhaltung von Zeitgarantien sichergestellt
  • CPU-Auslastung vorhersagbar
– probabilistische Planung (SPN, SRTF, HRRN):
  • Prozesszeiten unbekannt und nur approximierbar
  • Keine Zeitgarantie möglich
  • CPU-Auslastung nicht vorhersagbar

– statische Planung:
  • Vor Betrieb des Prozesses
  • Keine Planung im laufenden Betrieb
  • Ergebnis der Vorberechnung ist kompletter Ablaufplan
  • Begrenzt auf strikte Echtzeitsysteme
– dynamische Planung:
  • während des Betriebs des Prozesses
  • Stapelsysteme, interaktive Systeme, verteilte Systeme
  • schwache und feste Echtzeitsysteme

– assymetrische Planung:
  • wichtig bei assymetrischen Multiproessorsystemen
  • optional auf symmetrischen Multiprozessorsystemen
  • ungleiche Verteilung mehrer Prozesse auf CPUs
– symmetrische Planung:
  • identische Prozessoren
  • Lastausgleich bei Verteilung von Prozessen auf CPUs

Bekannte Verfahrensweisen:
– FCFS (First Come First Serve):
  • Prozesseinplanung nach Reihenfolge der Ankunft
  • Prozesse mit langen Rechenstößen werden begünstigt
  • Prozesse mit kurzen Rechenstößen werden benachteiligt
  • −: Konvoieffekt (Abwechseln kurze/lange Prozesse)
– RR und VRR: (Round Robin)
  • Verdrängende Variante von FCFS
  • Verringerung der Benachteiligung aus FCFS
  • Prozesse werden nach ihrer Ankuftszeit eingeplant
  • Prozessumplanung in regelmäßigen Zeitabständen
  • Durch Zeitschreibe entsteht CPU-Schutz
  • Weiterhin Problem des Konvoieffekts
  • VRR: Variable Zeitscheibe, nicht voll-verdrängend
  • VRR: Zusätzlich Vorzugsliste für IO
– SPN , HRRN , SRTF: (Shortest Process next)
  • SPN: Schnellster Prozess als nächstes
  • SPN: Prozessverhungern möglich
  • HRRN: Hungerfreies SPN durch Alterungswichtung
  • SRTF: Prozesseinplanung nach Bedienzeit
  • SRTF: unregelmäßige Prozessumplanung
  • SRTF: Verdrängt wird in eine Bereitliste
– MLQ, MLFQ:
  • Kombination der obrigen Strategien
  • Prozesseinplanung nach Typ und Ankuftszeit
  • Unregelmäßige Prozessumplanung
  • Jede Queue hat eigene Einplanungstrategie
  • Gut für kurze Prozesse, schlecht für lange

Prozesssynchronisation

Grundbegriffe:
– Kausalität: Beziehung zw. Ursache und Wirkung
– Gleichzeitige Prozesse:
  • vertikal: time sharing (multiplexen)
  • horizontal: multiprocessing
– Gekoppelte Prozesse: Zugriff auf gleiche Daten
– Sequentialisierung:
  • Koordinierung durch atomare Operationen
– unsicherer Zustand:
  • früher oder später kommt es zu Verklemmung
– Einseitige Synchronisation:
  • Nur ein beteildigter Prozess
  • erfolgt logisch oder bedingt
– Mehrseitige Synchronisation:
  • alle der beteildigten Prozesse betroffen
  • blockierend oder nicht-blockierend möglich

Nebenläufigkeit:
– Keine Abhängigkeit zw. Ereignissen
  • Datenabhängigkeit darf nicht verletzt werden
  • Zeitbedingung anderer darf nicht verletzt werden
– Ursachen:
  • Interrupts / preemptives Scheduling
  • Mono- Multiprozessor-Threads / Unix-Prozess-Signale

Verklemmung:
– Deadlock: gutartig, Prozesse sind blockiert
– Livelock: bösartig, Prozesse zw. laufend und bereit
– notwendige Bedingung:
  • 1. wechselseitiger Ausschluss
  • 2. Nachforderung eines oder mehrerer Betriebsmittel
  • 3. Unentziehbarkeit der zugeteilten Betriebsmittel
– notw. und hinreichende Bedingung:
  • 4. zirkuläres Warten
  • reicht vorzubeugen um Dealocks zu vermeiden
  • kann mittels Virtualisierung realisiert werden

Philosophenproblem:
– 5 Philosophen 5 Gablen
– Philosoph benötigt 2 Gabeln zum Essen
– Alle nehmen gleichzeitig linke Gabel
– Alle Philosophen verhungern

Blockierende Synchronisation:
– Locking Techniken wie Semaphore
– Kritische Abschnitte, vor welchen blockiert wird
– −: Verklemmung, Effizienzverlust

Nicht-blockierende Synchronisation:
– Keine kritischen Abschnitte
– Verwendung von atomaren Operationen
  • Prozessor-Befehle wie Compare-and-Swap
  • nebenläufigkeitsfreie Variablenmodifikation ohne Locks
– −: hohe Komplexität und Verhungern möglich

Semaphoren:
– Krit. Abschnitt mit wechselseitigem Ausschluss sichern
– Eignen sich für ein- und mehrseitige Synchronisation
– Ganzzahlige Variable mit zwei Operationen:
  • P : wait();
  • V : release();
– Operationen sind logisch und physisch unteilbar
– Auf P und V maximal ein Prozess gleichzeitig:
  • Prozesse werden in Warteschlangen gehalten
  • Nichtpassierende Prozesse werden schlafengelegt
– Atomarität wird auf Befehlssatzebene umgesetzt
  • nicht durch "normale" C-Anweisungen umsetzbar
– Mutex: eine Semaphorenspezialisierung
  • Prüft Eigentümerschaft bei Freigabe
  • lässt Freigabe damit bedingt zu
  • für einseitige und mehrseitige Synchro. geeignet

Monitor:
– Monitore sind abstrakte Datentypen
– Monitor Behandelt Parallelitätsprobleme automatisch
– Bei Blockierung muss Prozess den Monitor verlassen
– Konzept:
  • Mehrere Warteschlangen (WS)
  • Prozesse warten immer außerhalb des Monitors
  • MonitorWS: Prozesse warten auf Monitoreintritt
  • EreignisWS:Prozesse warten auf Wartebedingungsende
  • Verwendet bei Blockierung Bedingungsvariablen
  • nicht blockierend: Vorrang Signalgeber
  • blockierend: Vorrang Signalnehmer
– Hansen-Methode (blockierend):
  • Programmier-Primitive: wait() / broadcast()
  • Signalisierung lässt Signalgeber den Monitor verlassen Nachdem er alle Signalnehmer bereit gesetzt hat
– Hoare-Methode (blockierend):
  • Programmier-Primitive: signal()
  • Signal 1: Versetzt laufenden Prozess ins Warten
  • Signal 2: Prozess aus Warteschlange wird fortgesetzt
  • Signalisierung lässt Signalgeber den Monitor verlassen und genau einen Signalnehmer fortsetzen (atomar)
– Mesa-Methode (nicht blockierend):
  • Programmier-Primitive: signal()
  • signal unterbricht laufenden Prozess nicht
  • signal verschiebt Prozess in Monitorwarteschlange
  • Signalisierung lässt Signalgeber im Monitor fortfahren nachdem bereit-setzen eines oder aller Signalnehmer

Betriebsmittelverwaltung

Betriebsmittelklassifikation:
– wiederverwendbare: (begrenzt)
  • Anforderung durch mehrseitige Synchronisation
  • Zusätzliche Unterscheidung: teilbar und unteilbar
  • Werden belegt, benutzt und freigegeben
  • Beispiel: CPU, RAM, GPU
– konsumierbare: (unbegrenzt):
  • Anforderung durch einseitige Synchronisation
  • Werden produziert, empfangen, benutzt und zerstört
  • Beispiel: Signale und Traps

Ziele:
– Durchsetzung der vorgegebenen Betriebsstrategien
– Optimale Realisierung in Bezug auf relevante Kriterien
– Wichtig: Keine Verhungern und keine Verklemmung:

Aufgaben:
– Buchführung über vorhandene Betriebsmittel
– Steuerung der Verarbeitung von Anforderungen
– Betriebsmittelentzug bei fehlerhaften Prozessen

Verfahrensweisen:
– Statisch:
  • Aufgabenbewältigung vor der Laufzeit
  • Risiko: suboptimale Betriebsmittelauslastung
– Dynamisch:
  • Aufgabenbewältigung während der Laufzeit
  • Risiko: Verklemmung abhängiger Prozesse

Speicherverwaltung: Adressräume

Reale Adressen:
– lückenhafter, wirklicher Hauptspeicher
– Die realen Adressen, von der Hardware gegeben
– Vorgegebene Adressen und Adressräume
Logische Adressen:
– lückenloser, wirklicher Hauptspeicher
– Getrennte reale Adressbereiche werden linearisiert
– Zweck:
  • Sicherheit, Virtualisierung, bessere Verwaltung
Virtuelle Adresse:
– lückenloser, scheinbarer Hauptspeicher
– entkoppelt von der Lokalität im Arbeitsspeicher
– Kann größer als der Arbeitsspeicher sein (paging)
– Adresse wird auf den Hauptspeicher abgebildet
– Dafür gibt es dann Abbildungstabellen
– Navigation zum Speicher durch Abbildungstabellen

Eindimensionaler Adressraum:
– Typische Seitengröße bei Seitenadressierung: 4096
– Adressaufbau: Seitennummer p und Offset o (Versatz)
– Zusätzlich wird eine Seitentabelle benötigt
– Adressbildung:
  • Mit p Adresse in Seitentabelle Suchen
  • p in Zahl umrechnen und damit Eintragsindex suchen
  • Bei mehrstufigen: p gleichmäßig Aufteilen
  • o wird so wie es ist weiterhin übernommen
Zweidimensionaler Adressraum:
– Aufteilung des Adressraums in Segmente
– Zweikomponenten Adresse:
  • Segmentname S und Adresse A mit p und o (siehe oben)
– Adressbildung (ohne p und o):
  • Mit Segmentname auf Segmenttabelle zugreifen
  • Tabellen Eintrag mit Adresse verrechnen
– Adressbildung (mit p und o):
  • Mit Segmentname auf Segmenttabelle zugreifen
  • Mit diesen Eintrag auf die Seitentabelle zugreifen

Private Adressräume:
– Illusion eigenes Adressraums für BS und Maschinenprogs.
– Spezialhardware verhindert ausbrechen aus Adressraum
– Sinn: strikte Isolation ganzer Adressräume

Seitentabelle Berechnung bei Seitenadressierung:
– Adressraum : (Speicherseite : Eintrag)
  • Jeder Seite muss adressiert werden
  • Speicherseite : Eintrag – Einträge pro Seite
  • Adressraum : Einträge pro Seite – Ergebnis

Seitendeskriptor:
– Von MMU vorgegebener Verbund von Attributen:
  • seitenausgerichtete reale Adresse
  • Schreibschutzbit / Präsensbit
  • Referenzbit / Modifikationsbit
– Seitendeskriptor des BS in shadow page table
Filedeskriptor:
– Prozesslokale Integerdatei
– Wird von einem Prozess verwendet
– Zugriff auf Dateien, Geräte, Pipes, Sockets
Segmentdeskriptor:
– Von MMU vorgegebener Verbund von Attributen:
  • Basis: Segmentanfang im Arbeitsspeicher
  • Limit: Segmentlänge als Anzahl der Granulate
  • Typ (Text, Daten, Stapel)
  • Zugriffsrechte (lesen, schreiben, ausführen)
  • Expansionsrichtung / Präsensbit

Speicherbereiche

Code-Segment:
– Auszuführenden Programmcode
Data-Segment:
– inizialisierte Globale und statische Variablen
Heap-Segment:
– Von malloc reservierter Speicher
Block-Storage-Segment (BSS):
– nicht inizialisierte globale/statische Variablen
Stack-Segment:
– lokale (Pointer-) Variablen
– Übergebene Argumente
– Platz für Zwischenergebnisse
– Rücksprungadresse / Frame-Pointer

Adressraumschutz durch Eingrenzung:
– Begrenzungsregister legen Adressbereich fest
– Dieser ist im physikalischen Adressraum
– Auf diesen werden Speicherzugriffe beschränkt

Seitennummer und Versatz berechnen:
– Bits der Seitengröße berechnen
– logische Adresse unterteilen:
  • Hintere Bits sind für Seitengröße
– beide Adressen mit 0er vorne Auffüllen

Platzierungsstrategien:

– Bitkarte:
  • für Hohlräume fester Größe
– Lochliste:
  • für Hohlräume variabler Größe

Zuteilungsverfahren:
– Alle folgenden sind Listenbasiert
– Bei allen kann externer Verschnitt auftreten
– Freispeicherverschmelzung: first-fit schneller als worst-fit
– worst-fit:
  • absteigend nach Lochgröße sortiert
  • Größtes passendes Loch wird gesucht
  • +: Suchaufwand ist klein
  • −: Speicherverschnitt ist groß
– best-fit:
  • aufsteigend nach Lochgröße sortiert
  • kleinstes passendes Loch wird gesucht
  • +: Speicherverschnitt ist klein
  • −: Suchaufwand ist groß
– first-fit: (nach Adresse sortiert)
  • Erstes passendes Loch nutzen
  • +: Suchaufwand ist klein
  • −: Speicherverschnitt ist groß
– next-fit: (nach Adresse sortiert)
  • Round-Robin-Variante von first-fit
  • Beginnt Suche beim zuletzt zugeteilten Loch

– Interne Fragmentierung:
  • Speicherblöcke nur teilweise befüllt
  • Ist (durch das Betriebssystem) unvermeidbar
  • Buddy: Entstehung durch das Aufrunden
  • Kein Zugriffsfehler bei Zugriff auf ungenutzen Bereich
– Externe Fragmentierung:
  • Angeforderte Größe zu groß für jedes Loch
  • Dadurch Zerstückelung eines Speicherraums
  • Verringerung durch Speicherblock-Verschmelzung
  • Auflösung durch Kompaktifizierung möglich

Buddie-Verfahren:
– **Datenblock kann vorinitialisiert sein!!!**
– Hier Beschreibung des Binären Buddyverfahrens
– Neuer Speicher wird zu nächster 2er Potenz aufgerundet
– 2^n wird in 2^n großem Block gespeichert!
– Passender Speicher wird itterativ gesucht und verwendet
– Ist kein Speicher vorhanden:
  • Halbieren des ersten größeren Speichers, bis er passt
– Aufeinanderfolgende Buddyadressen um ein Bit different
– Lochlisten Einträge:
  • Links mit Zweierpotenzen sind Blockgrößen
  • Daneben: Adresse/n mit freiem Block jeweiliger Größe
– Bemerkungen:
  • Es gibt keinen externen Verschnitt

1. p1 = Malloc(200) / 2. p2 = Malloc(128) / 3. p3 = Malloc(500) / 4. free(p1) / 5. free(p2)

<table>
<tr><td colspan="4">1024</td></tr>
<tr><td colspan="3">512</td><td>512</td></tr>
<tr><td>p1</td><td colspan="2">256</td><td>512</td></tr>
<tr><td>p1</td><td>p2</td><td>128</td><td>512</td></tr>
<tr><td>p1</td><td>p2</td><td>128</td><td>p3</td></tr>
<tr><td>256</td><td>p2</td><td>128</td><td>p3</td></tr>
<tr><td>256</td><td colspan="2">256</td><td>p3</td></tr>
<tr><td colspan="3">512</td><td>p3</td></tr>
</table>

Ladestrategien:

– Umsetzung durch MMU (Memory Management Unit)
– Der TLB (Tranlation Lookaside Buffer) untersützt MMU
  • spezieller Cache mit Infos vom Seitendeskriptor
  • Speichert Ergebnis: Logisch – physikalisch

Arbeitsweisen:
– Demand-Paging:
  • Läd Adresse erst nach Ansprechen in den RAM
  • Nichtbenutzte Seiten werden ausgelagert
– Anticipatory:
  • Vorausladen bzw. Prefetching
  • Zur Vermeidung von Folgefehler
  • Heuristik liefert Hinweise über zukünftige Zugriffe

Fehler:
– Page/Segment-fault:
  • Speicher/Segment nicht im Hauptspeicher
  • Entstehen z.b. durch ungültige Zeiger
  • Present-Bit des Adressraums ist nicht gesetzt
  • Adresse ist somit vmtl. ausgelagert (Festplatte)
  • MMU löst einen Trap aus
  • Zugriffsfehler: Schwerer Seitenfehler

Ersetzungsstrategien:

Lokale vs. globale Seitenersetzung:
– lokal:
  • Suchraum: Menge residenter Seiten des Prozesses
  • ein Seitenfehler ist vorhersag-/reproduzierbar
– global:
  • Suchraum: Menge alles residenter Seiten aller Prozesse
  • ein Seitenfehler ist unvorhersag-/unreproduzierbar

Lokalitätsprinzip:
Wenn ein Prozess eine Stelle in seinem Adressraum referenziert, dann wird er wahrscheinlich dieselbe Stelle oder eine andere Stelle in direkter Umgebung referenzieren

Seitenflatter:
– Seitenein-/auslagerungen dominiert Systemaktivität
– Sofortiges Einlagern kürzlich ausgelagerter Seiten
– ein mögliches Phänomen der globalen Seitenersetzung
– Verschwindet meist von alleine wieder

Freiseitenpuffer:
– Um schneller einen freien Seitenrahmen zu finden
– FIFO mit Cache für potentiell zu ersetzende Seiten
– Seiten im Cache werden wie bei SSD überschrieben
– Zugriffsfehler auf Seite im Cache:
  • Reklamierung und Neusetzen des Präsensbits

Arbeitsmenge:
Die kleinste Sammlung von Programmtext und -daten, die in einem Hauptspeicher vorliegen muss, damit effiziente Programmausführung zugesichert werden kann.

Strategien:
– FIFO (First in First out):
  • Ersetzt wird zuletzt eingelagerte Seite
  • Implementierung: Verkettete Liste
  • Mehr Seitenrahmen führen zu mehr Seitenfehlern
– LFU (Least frequently used):
  • Ersetzt wird die am seltensten referenzierte Seite
  • Implementierung durch Zähler
– LRU (Least recently used):
  • Ersetzt wird am längsten nicht referenzierte Seite
  • Siehe: Verschiedenes

Dateisysteme:

Kontinuierliche Speicherung:
– Dateispeicherung in aufsteigend nummerierten Blöcken
– Zur Identifizieren: Anfangsblock und Größe
– +: Schnelles Lesen, da keine externe Fragmentierung
– +: Geeignet für Echtzeitsysteme
– +: Freien Speicher finden ist schwierig
– −: Interne Fragmentierung
– −: dynamische Erweiterung sehr aufwändig

Verkettete Speicherung:
– Speicherung von Daten in verketteten Blöcken (Liste)
– +: Dynamische Erweiterung einfach möglich
– −: hohe Fehleranfälligkeit / externe Fragmentierung

Indiziertes Speichern:
– Speicher in gleich große Blöcke aufgeteilt
– Pro Datei ein (evtl. mehrfach verketteter) Inode der auf die Datenblöcke in der richtigen Reihenfolge zeigt.
– Extents: Indexblock zeigt auf Datenblock aber diese Daten sind Pointer auf anderen Speicher

Freispeicherverwaltung:
– Speicherbereich wird in gleich Große Blöcke unterteilt
– Bitvektor speichert pro Block belegt / nicht belegt
– verkettete Liste repräsentiert freie Blöcke
– Besser: Aufeinanderfolgende Blöcke Zusammenfassen
– Besser: Statt Liste: Datenblöcke zeigen auf freien Speicher

Unix Blockstruktur:

Bootblock	Superblock	Inodeliste	Datenblock

– Boot: Infos zum Laden eines initialen Programmes
– Super: Anzahl Blöcke / Inodes und Attribute
– Inode: Dateiart,UserId,rights,größe,linkzähler,hard links
  • Zusätzlich Zeiten: Erstellung, Zugriff, letzte Änderung

Journaling-File-Systems:
– Änderungen als Transaktionen dokumentiert im Log-File
– Log wird vor der Transaktion beschrieben
– Inkonsistenz wird beim Booten durchs Log-File vermieden
– −: Undo nach erkanntem Transaktionsproblem
– Optimiert: Checkpoint (Platte in konsistentem Zustand)
  • Log-File-Einträge bis checkpoint löschen

Raid 0: Gestreifte Platte (Paritätsfrei):
– Daten werden über mehrere Platten gespeichert
– +: Schnelles Lesen und schreiben
– −: Systemausfall bei Plattenausfall / keine Fehlertoleranz

Raid 1: Gespiegelte Platten (Paritätsfrei)
– Zwei oder n Platten mit gleichen Daten
– Verknüpfung von Raid 0 und Raid 1 möglich
– +: n-1 Platten können ausfallen
– +: Schnelles Lesen (Nicht Schreiben)
– −: Hoher Speicherbedarf

Raid 4: Paritätsplatte:
– Daten werden über mehrere Platten verteilt gespeichert
– Eine Paritätsplatte P speichert Paritätsinformationen
– +: Eine Platte kann ausfallen und schnelles Lesen
– +: mind. 3 Platten, beliebig viele mehr möglich
– −: P hoch belastet, da bei jedem Schreiben einbezogen

Raid 5: Verstreuter Paritätsblock:
– Paritätsblöcke werden über alle Platten verteilt
– Vor und Nachteile wie Raid 4
– +: Last der Paritätsplatte wird verteilt
– Raid 6: zwei Paritätsblöcke pro Platte

Verschiedenes:

Hacking:

– strcpy() und strcat() als Sicherheitslücken

Least Recently Used:

– Periodische Unterbrechung – Hintergrundrauschen
– Altersstruktur:
  • Benötigt in Software implementiertes Schieberegister
  • Zusätzlich Zeitgeber für periodische Unterbrechung
  • Referenzbitprüfungs nach jedem Zeitintervall
  • Ersetzt: Global älteste Seite im Aging Register

tick	Ref.	aging register
		00000000
n	1	10000000
n+1	0	01000000
n+2	1	10100000

– Zweite Chance:
  • Basiert auf FIFO, um ein billiges LRU nachzubauen
  • Nutzt Zeitintervall zur periodischen Unterbrechung
  • Ersetzung der Ersten Seite bei der Referenzbit 0 ist
  • Wenn alle Referenzbits 1: FIFO Prinzip nutzen!
– Dritte Chance:
  • Übernimmt alle Merkmale von Second Chance
  • Zusätzliche Nutzung des Modifikationsbits
  • (0,0): unbenutzt = beste Wahl
  • (1,0): beschrieben = Naja
  • (0,1): gelesen = Naja
  • (1,1): kürzlich beschrieben = Schlecht
  • Zwei Umläufe für jede eingelagerte Seite

Wertetabelle:

Hex	Dez	Binär	Name	Name	2er	Dez	Pot
0	0	0000	KiB	KibiB	2^{10}	2	2^1
1	1	0001	MiB	MibiB	2^{20}	4	2^2
2	2	0010	GiB	GibiB	2^{30}	8	2^3
3	3	0011	TiB	TebiB	2^{40}	16	2^4
4	4	0100	PiB	PebiB	2^{50}	32	2^5
5	5	0101				64	2^6
6	6	0110				128	2^7
7	7	0111				256	2^8
8	8	1000				512	2^9
9	9	1001				1024	2^{10}
A	10	1010	Name	10er	2er	2048	2^{11}
B	11	1011	kB	10^3	2^{10}	4096	2^{12}
C	12	1100	MB	10^6	2^{20}	8192	2^{13}
D	13	1101	GB	10^9	2^{30}	16384	2^{14}
E	14	1110	TB	10^12	2^{40}	32768	2^{15}
F	15	1111	PB	10^15	2^{50}	65536	2^{16}

Gewichtsklasse von Prozessen:

– Schwergewichtiger Prozess:
  • Eigener Adressraum und Ausführungsfaden
– Leichtgewichtiger Prozess:
  • teilt sich Adressraum mit anderes Prozessen
– Federgewichtiger Prozess:
  • weder eigener Adressraum noch Ausführungsfaden
  • wird nicht vom Kernel verwaltet

Kleinigkeiten zum Einfügen:

Inode-Aufgaben:

Gegeben:
/Desktop/Inode:

71	d...	3	...	4096	...	.
66	d...	9	...	4096	...	..
58	d...	2	...	4096	...	deeper
27	l...	1	...	53	...	file.txt – deeper/file.txt
93	-...	1	...	25	...	prog.c

/Desktop/Inode/deeper:

58	d...	2	...	4096	...	.
71	d...	3	...	4096	...	..
94	-...	1	...	6	...	file.txt

Lösung:
– Inode mit: Ordner, Datei, Verknüpfung
– Ordner: Inhalt + st_ino + . + ..
– Datei: Inhalt unbekannt: ???
– Verknüpfung: Verknüpfte Datei
– Alle Inode (st_ino) Nummer müssen abgearbeitet werden
– Jede Inode Nummer nur einmal

Inode			Inhalt Datenblöcke	
st_ino:	71		71 .	66 ..
st_nlink:	3	⟶	58 deeper	27 file.txt
st_size:	4096		93 prog.c	

st_ino:	66		66 .	71 Inode
st_nlink:	9	⟶		
st_size:	4096			

st_ino:	58		58 .	71 ..
st_nlink:	2	⟶	94 file.txt	
st_size:	4096			

st_ino:	27		
st_nlink:	1	⟶	deeper/file.txt
st_size:	53		

st_ino:	93		???	
st_nlink:	1	⟶		
st_size:	25			

st_ino:	94		???	
st_nlink:	1	⟶		
st_size:	6			

Bemerkung:
– Root Knoten zeigt mit . und .. auf sich selbst
– Inode auf Datei wird getrennt von Datei gespeichert

UNIX-UFS-Dateisystem:

– Pfadnamen ohne '/' am Anfang:
  • Werden relativ zum aktuellen Verzeichnis interpretiert

– Hierarchisch organisierter Namensraum:
  • Die Absteigende Baumstruktur bei Dateisystemen
  • Kontext (Verzeichnis) als flachen Namensraum
  • Gleiche Namen in unterschiedlichem Kontext möglich
  • Pro Verzeichnis mehrmals gleicher Name Nicht möglich

Dateisystem:
– Symbolic-Links:
  • Können existieren, obwohl Ziel bereits gelöscht
  • Kann auf Verzeichnisse verweisen
– Hard-Link:
  • Pro Reguläre Datei mind. einer im selben Dateisystem
  • Pro Verzeichnis mind. 2 Hard-Links
  • Können nur auf Dateien verweisen
  • Auf Datei: Anlegen nur im selbem Dateisystem

Prozesskontrollblock:
– Als Tabelle vorstellbar mit folgenden Einträgen:
  • Zustande des Prozesses / Befehlszähler
  • CPU-Register / Stack-Pointer
  • Zustande der geöffneten Dateien / aktuelles Verzeichnis
  • Verwaltungsinformationen / UID Eigentümer

SP - Ausschnitte

25.07.2023 - Systemprogrammierung

1. Inhaltsverzeichnis

2. Definitionen

Programm: Folge von Anweisungen, kann von mehreren Prozessen gleichzeitig ausgeführt werden

Prozess: Ein in Ausführung befindliches Programm (und nur genau eines)

Compiler: Erzeugt aus mehreren Programmteilen (Modulen) ein Programm (hier genau auf die Formulierung achten!)

Binder: Erzeugt aus einer oder mehreren Objekt-Dateien ein Programm (laut Pad)

Prozessorkern: Pro Prozessorkern maximal einen laufenden Prozess

3. Adressraum

- Aufteilung des Hauptspeichers eines Prozesses in Segmente
  - Vgl. Vorlesung *A-III, Seite 7f.*

```
static int a = 3; static int b;
static int c = 0; const int f = 42;
const char *s = "Hello World\n";

int main(void) {
    int g = 5;
    static int h = 12;
}
```

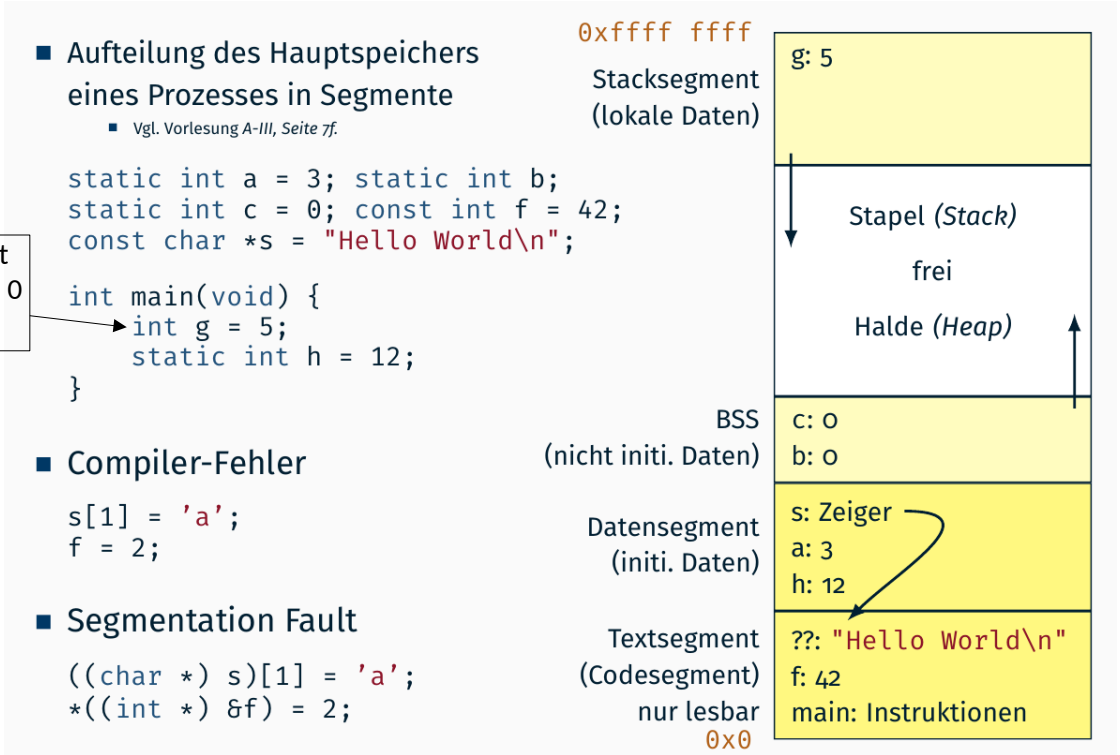
Wird auch nicht automatisch mit 0 initialisiert

- Compiler-Fehler

```
s[1] = 'a';
f = 2;
```

- Segmentation Fault

```
((char *) s)[1] = 'a';
*((int *) &f) = 2;
```



Stack-Frame

Alles was beim Funktionsaufruf und durch die Funktion selbst auf den Stack gelegt wird (Übergabe-Parameter, Rücksprungadresse, lokale Variablen, Caller-Save-Register).

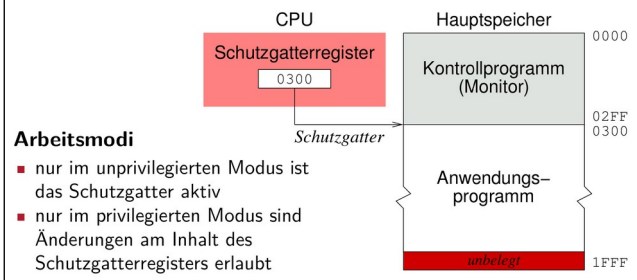
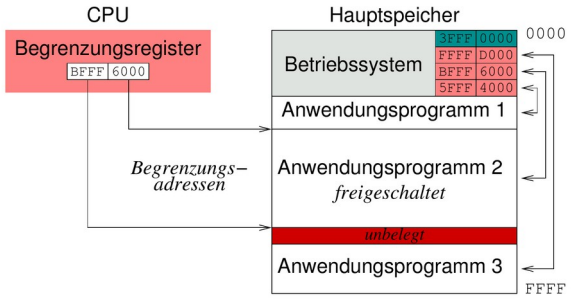
Beim Rücksprung wird dieser Stack-Frame wieder entfernt.

Makros bspw. #define sum(a,b) as a + b

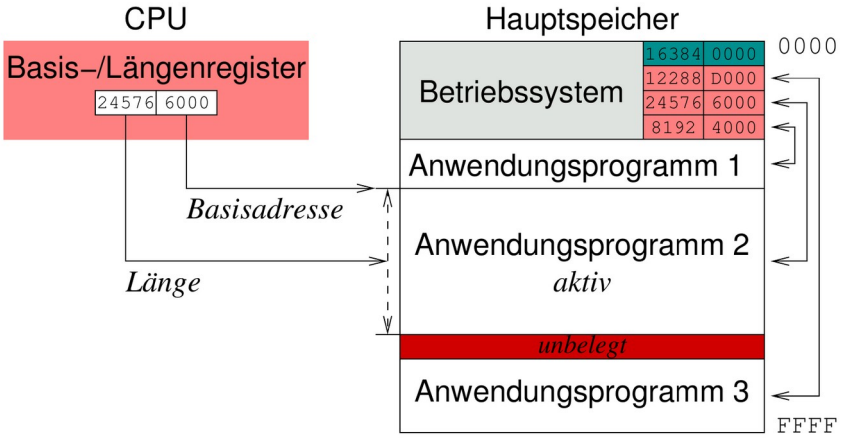
Präprozessor ersetzt einfach das, was links vom *as* steht durch das was rechts vom *as* steht

→ bei Sum oderso wird nicht die Reihenfolge garantiert: sum(4,5) * 3 = 4+5*3 = 19 nicht 9*3 =27

3.2 ADRESSRAUMSCHUTZ

Adressraumschutz durch Abgrenzung/Abteilung (Fencing)	**Adressraumschutz durch Einfriedung/Eingrenzung**
	
Problem: Externe Fragmentierung	Problem: Externe Fragmentierung

Adressraumschutz durch Segmentierung



Problem: Pro Programm nur ein Segment

3.3 Adressraumarten

Virtueller Adressraum

- Abstrahiert ganze Prozesse im Arbeitsspeicher (Swapping möglich)
- Erlaubt es ein Programm laufen zu lassen, obwohl nur Einzelbestandteile von Text und Daten eingelagert sind
- Kann größer sein, als der physikalische Speicher

Logischer Adressraum

- Abstrahiert einzelne Speicherbausteine im Arbeitsspeicher
- Wird vor/zur Laufzeit auf reale Adresse abgebildet
- Lückenloser richtiger Hauptspeicher (gleich groß wie phys.), es gibt aber trotzdem „fake“ Adressen

Paging

Segmentierung

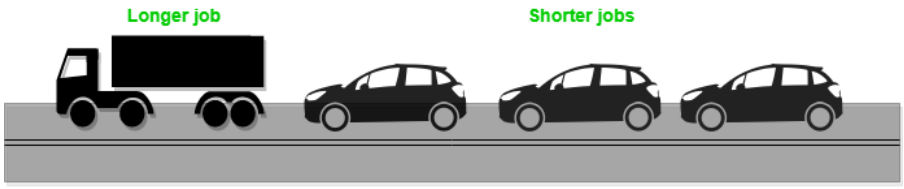
Physikalischer Adressraum

Realer Speicher mit Lücken, etc.

4. Prozesseinplanung (Ausschnitt)

4.1 Konvoi-Effekt

Längere Prozesse zögern kürzere Prozesse unnötig hinaus.



4.2 Round Robin

FCFS aber mit Verdrängung: Jeder Prozess bekommt Zeitscheibe die er arbeiten darf, periodische Umplanung der Prozesse, damit kurze Prozesse früher dran sind

→ Immernoch etwas Konvoieffekt weil kurze Prozesse ihre Zeitscheibe nicht ausnutzen aber besser

Virtual Round Robin

Kürzere Prozesse (die ihre Zeitscheibe nicht voll ausgenutzt haben) werden gezielt bevorzugt. Z. B. mit variabler (kürzerer) Zeitscheibe, schnellerer Verdrängung oder früheren Plätzen in der (FCFS) Warteliste.

4.3 Echtzeitbetriebssystem

Eigenschaften

- Einhaltung vorgegebener Termine
- Rechtzeitigkeit anstelle von Geschwindigkeit
- Vorhersagbarkeit
- Überwachung / Interaktion mit physikalischer Welt

Verhaltensunterschiede bei Terminvorgaben

Feste Terminvorgaben (firm) – Terminverletzungen tolerierbar

- Abbruch der Berechnung (durch Betriebssystem) → Ergebnis wertlos
- Start der nächsten Berechnung (durch BS)
- Transparent für die Anwendung

Harte Terminvorgaben (hard) – Terminverletzung keinesfalls tolerierbar

- Betriebssystem löst Ausnahmesituation aus
- Ausnahmebehandlung führt System in sicheren Zustand
- Intransparent für die Anwendung

4.4 Kriterien für die Prozesseinplanung

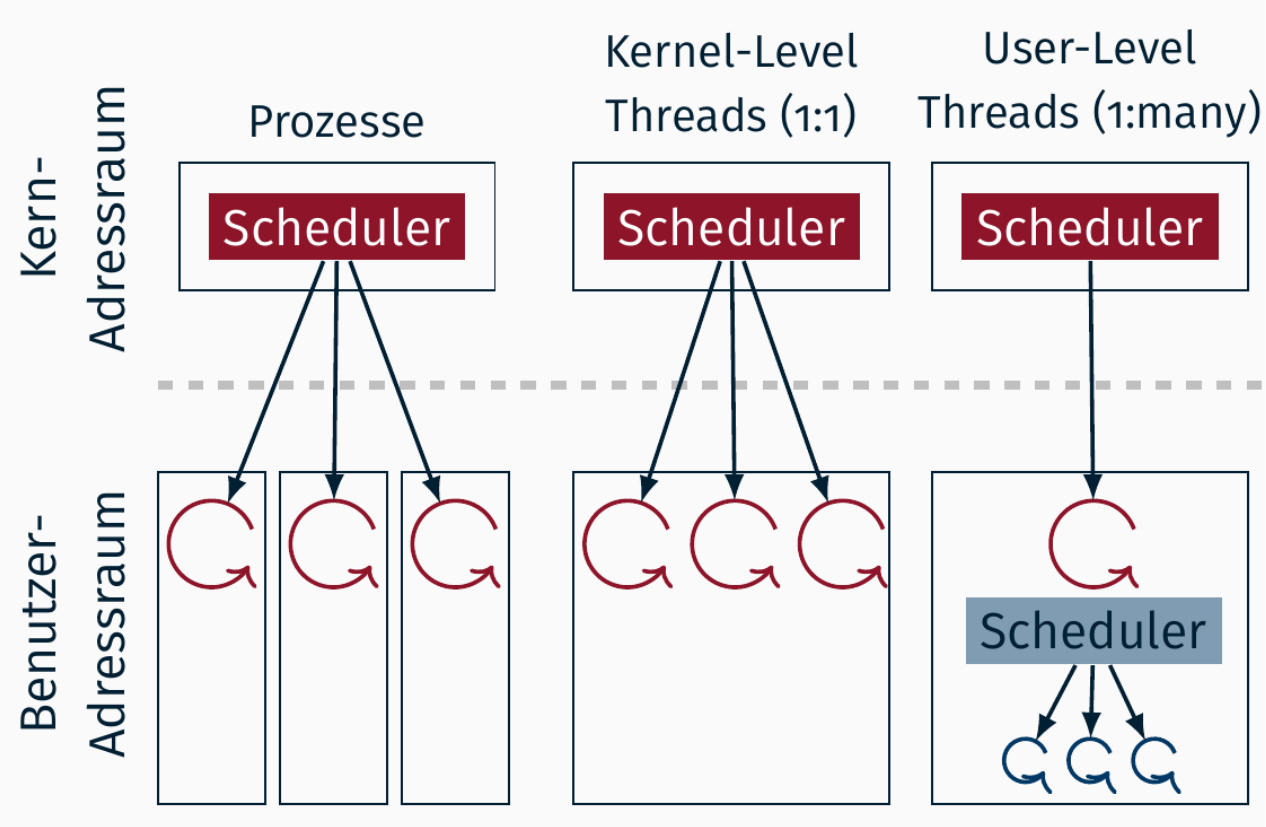
Benutzerorientiert	Systemorientiert
Das vom Nutzer wahrgenommene Verhalten Bspw. eine möglichst kurze Antwortzeit bei System-aufrufen, DurchLaufzeit des ges. Prozesses	Eine effektive Auslastung von Betriebsmitteln Bspw. ein möglichst hoher Durchsatz an vollendeten Prozessen, gleichmäßige Prozessauslastung
– Antwortzeit – Durchlaufzeit (Zeit vom Start bis zur Beendigung von einem Prozess minimieren) – Termineinhaltung – Vorhersagbarkeit	– Durchsatz – Prozessorauslastung – Gerechtigkeit – Dringlichkeit – Lastausgleich

Prioritäten je nach Betriebsart:

Allgemein	Stapelbetrieb	Dialogbetrieb	Echtzeitbetrieb
Gerechtigkeit, Lastaus-gleich	Durchsatz, Durchlaufzeit, Prozessorauslastung	Antwortzeit	Dringlichkeit, Termin-einhaltung, Vorhersagbar-keit
			Konflikte: Gerechtigkeit, Lastausgleich

5. MULTI-THREADING

Prozesse	Kernel-Level-Threads (Schwergewichtig)	User-Level-Threads (Leichtgewichtig)
• Ausführungsumgebung mit einem Aktivitätsträger • Keine parallelen Abläufe innerhalb eines logischen Adressraums auf Multi-prozessorsystemen	• Teurer als User-Threads • Gruppe von Threads nutzt gemeinsam die Betriebs-mittel eines Prozesses • Schedulingstrategie vom System vorgegeben • Systemkern sieht jeden Kernel-Thread als eigenen Aktivitätsträger: → Multi-Threading	• Extrem billig zu erzeugen • Schedulingstrategie vom Programmierer vorgege-ben • Realisierung auf Anwen-dungsebene • Systemkern sieht nur ei-nen Kontrollfluss: → Ein blockierender User-Thread blockiert alle User-Threads → Kein Multi-Threading
fork();	*pthread_create();* *(pthread_detach();* Wenn der Rückgabewert egal ist und der Speicher sofort nach Be-endigung freigegeben werden soll)	



5.2 Probleme bei FORK und EXEC:

- Nur der aufrufende Thread wird geklont (alle anderen Threads sind im Child nicht mehr vorhanden)
- Gelockte Mutexe bleiben gelockt und können nicht freigegeben oder zerstört werden
- Kind kann inkonsistenten Zustand kopieren
- Dateideskriptoren werden bei `fork();` vererbt, und bei `exec();` auch nicht geschlossen =>

```
int fd = open("index.html", O_RDONLY | O_CLOEXEC);
FILE *fp = fdopen(fd, "r");
```

Aber: Bei *exec* werden alle Mutexe zerstört und alle weitere Threads verschwinden

5.3 Volatile

Variable wird immer neu aus Speicher geholt und nicht lokal aus Cache oder so

→ Man bekommt Änderungen mit, aber nur an dieser Variable (vgl Java)

5.4 Verklemmungen (Deadlock)

Eine irreversible gegenseitige Blockierung von Prozessen, bei der das Prozessssystem keinen Fortschritt mehr macht (still steht).

Bedingungen

Notwendig:

1. Ausschließlichkeit der Betriebsmittelnutzung (mutual exclusion)
2. Nachforderung von Betriebsmitteln (hold and wait)
3. Unentziehbarkeit der Betriebsmittel (no preemption)

Hinreichend & Notwendig:

4. Zirkuläres Warten auf die Betriebsmittel

→ Für eine Verklemmung müssen alle 4 erfüllt sein

5.5 Verklemmungsvorbeugung (Deadlock prevention)

- Nicht-blockierende Synchronisation
- Alle benötigten Betriebsmittel atomar anfordern
- Betriebsmittel virtualisieren → Die realen Betriebsmittel können entzogen werden
- Betriebsmittel in fester Reihenfolge zuteilen

5.6 Verklemmungsvermeidung (Deadlock avoidance)

- Nur zirkuläres Warten wird durch laufende Analyse verhindert
- Vor jeder Betriebsmittelanforderung wird geprüft, ob System in einem unsicheren Zustand wäre

Bankiersalgorithmus

- System weiß genau welcher Prozess welche Betriebsmittel anfordern will/schon hat (unrealistisch). Der Algorithmus kann für so einen Fall Testen ob Deadlocks auftreten würden.
- → Falls ja wird die Anforderung der Mittel zurückgewiesen

Betriebsmittelgraph

- System weiß welche Betriebsmittel ein Prozess anfordert und erstellt daraus einen Graphen
- Durch Analyse dieses Graphen können Zyklen erkannt werden

5.7 Nicht-blockierende Synchronisation (CAS)

Vorteile	Nachteile
Rein auf Anwendungsebene, keine teuren Systemaufrufe Geringere Mehrkosten als bei Locking, wenn die CAS-Operation auf Anhieb funktioniert Konkurrierende Threads werden vom Scheduler nach dessen Kriterien eingeplant Keine Abhängkeit vom Halter eines Locks Verklemmungsfrei	Schwere algorithmische Umsetzung Verhungern (komplexe Änderung kann immer wieder von kürzerer Änderung ungültig gemacht werden) Braucht Hardware-Unterstützung

Wird v.a. bei „**optimistischen Ansätzen**“ verwendet, bei denen davon ausgegangen wird, dass Prozesse nicht gleichzeitig eintreffen, es also wenig Konkurrenz gibt. Falls es doch dazu kommt werden Konflikte nachträglich behandelt mittels hardwaregesteuertem wechselseitigem Ausschluss (CAS, TAS, FAA).

5.8 Sonstiges

Bei Monoprozessorsystemen ohne Verdrängung braucht man logischerweise keine Synchronisation

Pessimistischer Ansatz: Vmtl. viel Konkurrenz zwischen Prozessen, deswegen blockierende Synchronisation im wechselseitigen Ausschluss

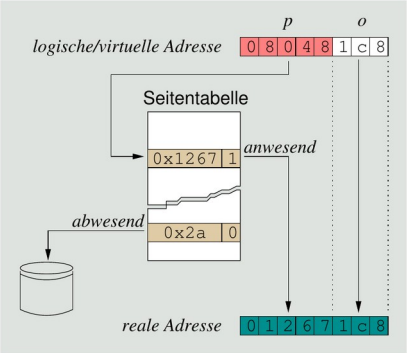
Mehrseitige Synchronisation: Alle beteiligten Prozesse sind betroffen

Einseitige Synchronisation: Nur einer der beteiligten Prozesse betroffen

Kategorien von Betriebsmitteln

Wiederverwendbare Betriebsmittel	Konsumierbare Betriebsmittel
Begrenzter Zugriff	Unbegrenzter Zugriff
Existieren dauerhaft (persistent)	Existieren vorübergehend (flüchtig)
Sind entweder teilbar oder unteilbar sein	Werden erzeugt und wieder zerstört
Beispiele in Hardware: CPU, RAM, GPU	Beispiele in Hardware: Signale, Traps
Beispiele in Software: kritischer Abschnitt, Variable	Beispiele in Software: Meldungen, Datenstrom

6. PAGING



6.1 SEITENDESKRIPTOR

Page Table: Seitennummer → Seitendeskriptor

Seitendeskriptor:

- Attribute: anwesend, referenziert, modifiziert, schreibbar, ausführbar
- Kachelnummer/-adresse im Hauptspeicher (falls anwesend, eingelagert) oder Blocknummer in der Ablage (falls abwesend, ausgelagert)

6.2 ABLAUF BEIM PAGE FAULT

1. MMU erkennt nicht gesetztes *present-bit* → Trap
2. Wenn Seite ungültig: SIGSEGV (Prozess in beendet)
3. Wenn Seite gültig: Seite wird angefordert und eingelagert (Prozess in blockiert):
   - Falls Seite im Freiseitenpuffer, muss Seite nicht vom Hintergrundspeicher geladen werden
   - Falls keine freie Seite im Arbeitsspeicher: Ungenutzte Seite auslagern (nach Seitenersetzungsstrategie)
   - Seite in freien Seitenrahmen einlagern und als *present* markieren (Prozess in bereit)
4. Prozess wird vom Scheduler in laufend überführt: Zugriff wiederholen

6.3 LEAST RECENTLY USED

Legt fest welche Page (?) aus dem Cache entfernt wird wenn er voll ist. Also hier die bei der der letzte Zugriff am längsten her ist. Grundsätlich immer mit FIFO Liste der Elemente

Zweite Chance (Basiert auf FCFS):

Used Bit welches periodisch auf 0 gesetzt wird und wenn es benutzt wird auf 1

→ Dann aus FIFO Liste in der alle drin sind das erste Element mit Used Bit = 0 entfernen

Dritte Chance:

Betrachte (Used Bit, Dirty Bit) wobei Dirty Bit: 1 falls Seite beschrieben wurde, wird nicht genullt

→ Dann am besten was mit (0,0) entfernen sonst (0,1) und dann (1,0) dann (1,1)

6.4 SONSTIGES

- Seitenflattern: Seitenein-/auslagerungen dominieren die Systemaktivität
- Freiseitenpuffer: Puffert freie Seiten damit man schneller eine freie Seite findet
- Demand-Paging: Page wird erst eingelagert wenn sie benötigt wird
- Anticipatory-Paging: Pages werden auch schon vorher (je nach Zugriffsmuster) geladen

6.5 Externe und Interne Fragmentierung

Interne Fragmentierung (Paging)	Externe Fragmentierung (Segmentierung)
• Alle Blöcke gleich groß • Innerhalb der Blöcke ist der Speicher nur teilweise befüllt → Kein Zugriffsfehler bei Zugriff auf ungenutzten Bereich	• Blöcke unterschiedlich groß • Zwischen den Blöcken ist Speicher ungenutzt

7. Speicherverwaltung

7.1 Dateisysteme – Gemischtes

Namensräume

- Namensraum: Baumstruktur bei der Objekte über Pfadnamen eindeutig angesprochen werden
- Flacher Namensraum: Nur ein Verzeichnis → Eindeutigkeit der Pfadnamen geht nur mit eindeutigen Objektnamen
- Hierarchischer Namensraum: Verschiedene Verzeichnisse → In verschiedenen Verzeichnissen gleicher Name erlaubt

Indiziertes Speichern:

- Speicher in gleich große Blöcke aufgeteilt
- Pro Datei ein (evtl. mehrfach verketteter) Index-Block (Inode) der auf die Datenblöcke in der richtigen Reihenfolge zeigt.
- Extents: Indexblock zeigt auf Datenblock aber diese Daten (die als Daten gelistet sind) sind Pointer auf anderen Speicher → neues Konzept, bei modernen Systemen teilweise eingeführt

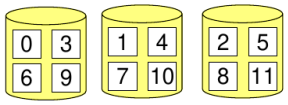
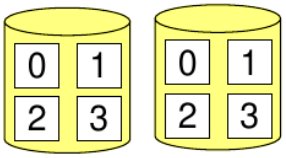
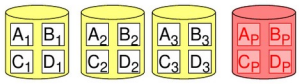
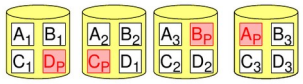
Journaling File-Systems

- Alle Änderungen an Daten & Meta-Daten werden vor der Änderung als Transaktionen (mit Informationen zum Rückgängigmachen und Wiederholen) protokolliert
- Wurden sie erfolgreich durchgeführt, wird dies (erst danach) auch protokolliert
- Bei einem Absturz kann das Dateisystem wieder in einen konsistenten Zustand gebracht werden
- Nach dem Erreichen bestimmter Checkpoints wird das Log-File gelöscht

Dateien löschen: Zugriffsrechte *read*, *write*, *exec* auf übergeordnetem Ordner notwendig aber nicht unbedingt auf Datei selbst!!

Dateideskriptor: Eine prozesslokale Integerzahl, die der Prozess zum Zugriff auf eine Datei, ein Gerät, einen Socket oder eine Pipe benutzen kann.

7.2 RAID

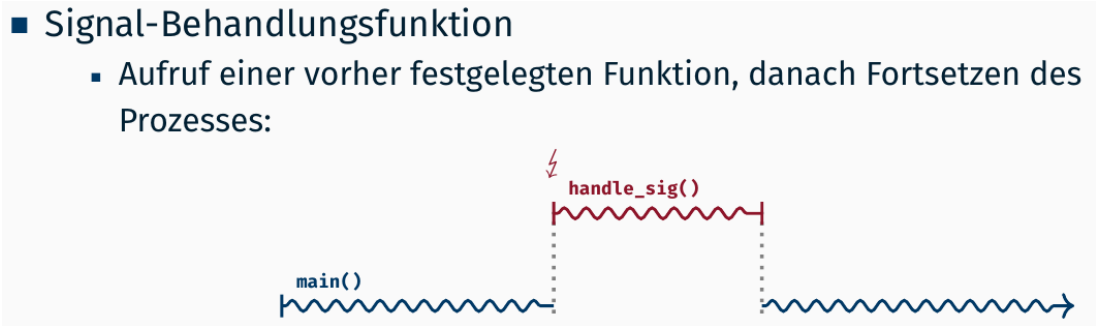
RAID 0	RAID 1	RAID 4	RAID 5
		 Paritätsblock ist byteweises XOR	 RAID 6: Doppelte Paritätsblöcke

Allgemein sind alle Verfahren schneller als einzelne Festplatten, da die Daten auf mehrere Platten verteilt sind.

7.3 Static- & Dynamic Binding

Static Libraries	Dynamic Libraries (Shared Libraries)
• Dateiendung: *.a* • Linker bindet alle .o-Dateien aus der Library, die bis dahin unaufgelöste „Commands“ enthalten (die Reihenfolge der Libraries kann somit eine Rolle spielen) • Library wird zur Ausführung des Programmes nicht mehr benötigt	• Dateiendung: *.so* • Code enthält relative Adressen/Platzhalter (Position-Independent Code, (`-fPIC`)) • Adressen werden erst beim Programmstart/ Laden gebindet (*dynamic linker*) • Vorteil: Speicherplatzersparnis & Updates

8. Signale



Es entsteht (nicht-echt-parallele) Nebenläufigkeit, bspw. mit der *errno*! (Der Signalhandler selber kann aber nicht unterbrochen werden und läuft immer komplett durch)

8.2 Wichtige Signale

- SIGINT: Ctrl+C
- SIGTERM: kill();
- SIGPIPE: Schreiben auf Socket, nachdem Gegenseite geschlossen wurde
- SIGSEGV: Segmentation Fault
- SIGABORT: abort();
- SIGFPE: Floating-Point-Exception (Überlauf, Geteilt durch 0)
- SIGCHLD: Child ist zum Zombie geworden

Definitionen:

Progr.: Folge von Anweisungen, kann von mehreren Prozessen gleichzeitig ausgeführt werden.

Prozess: Ein (!) in Ausführung befindliches Progr.

Compiler: Erzeugt aus mehreren Programmteilen (Modulen) ein Progr.

Binder: Erzeugt aus Objekt-Dateien ein Progr

Multi-Threading

Prozesse: Ausführungsumgebung mit einem Aktivitätsträger
fork();

Kernel-Level-Threads:
- Teurer als User-Level,
- Schedulingstrategie vom BS vorgegeben
- Systemkern sieht jeden Thread als eigenen Aktivitätsträger → Multi-Threading
- Gruppe v. Threads nutzt gemeinsame Betriebsmittel eines Prozesses pthread

User-Level-Threads:
- Billig
- Schedulingstrategie vom Programmierer vorgegeben
- Realisierung auf Anwendungsebene
- Systemkern sieht nur einen Kontrollfluss
  ↳ Kein (echtes) Multi-Threading
  ↳ Ein blockierender User-Thread blockiert alle User-Threads

Verklemmung

Eine irreversible Blockierung von Prozessen, bei der das Prozesssystem keinen Fortschritt mehr macht, bzw. still steht.

Notwendige Bed.:
- Ausschließlichkeit der Betriebsmittelnutzung
- Nachforderung von Betriebsmitteln
- Unentziehbarkeit der Betriebsmitteln

Hinreichende & notw. Bed.:
- Zirkulläres Warten auf Betriebsmittel

Verklemmungsvorbeugung

- Nicht blockierende Synchronisation
- Alle benötigten Betriebsmittel auf einmal atomar anfordern
- Betriebsmittel virtualisieren (→ reale können entzogen werden)
- Betriebsmittel in fester Reihenfolge zuteilen

Verklemmungsvermeidung

- Nur zirkuläres Warten wird durch laufende Analyse verhindert
- Vor jeder Betriebsmittelanforderung wird geprüft, ob System in unsicherem Zustand
  ↳ Bankiersalgorithmus, falls ja wird Anforderung zurückgewiesen

Betriebsmittelgraph: Zyklus → Verklemmung

Nicht-blockierende Sync.

Vorteile:
- Rein auf Anwendungsebene, keine teuren Systemaufrufe
- Geringere Mehrkosten als bei Locking, wenn CAS auf anhieb funktioniert
- Konkurrierende Threads werden vom Scheduler nach dessen Kriterien eingeplant
- Keine Abhängigkeit vom Halter d. Locks
- Verklemmungsfrei

Nachteile:
- Schwere algorithmische Umsetzung
- Verhungern (wenn lange Berechnung)
- Hardware-Unterstützung

Optimistischer Ansatz:
- Wenig Konkurrenz
- Falls doch Konflikt wird mit hardware-gesteuerten wechselseitigen Ausschluss behandelt (CAS)

Pessimistischer Ansatz:
- Viel Konkurrenz → Block. Sync.

Bei Monoprozessor ohne Verdrängung keine Synchronisation

Betriebsmittel:

Wiederverwendbar: Begrenzte Zugriff
Existieren dauerhaft (persistent)
Sind entweder teilbar e. unteilbar
Bsp. in Hardware: CPU, RAM, GPU
Bsp. in Software: Krit. Abschnitt

Konsumierbar: Unbegrenzter Zugriff
Existieren vorübergehend (flüchtig)
Werden erzeugt & wieder zerstört
Bsp. in Hardware: Signale, Traps
Bsp. in Software: Datenstrom

Prozesseinplanung

Kooperativ	Präemptiv
Kein CPU-Entzug ↳ CPU-Monopolisierung FCFS, Batch-Betrieb	CPU-Entzug RR, Mehrprgr.-Betrieb
Deterministisch	Probabilistisch
Prozesszeiten bekannt ↳ Zeitgarantien	Prozesszeiten unbek. Shortest Process Next
Statisch	Dynamisch
Kompletter Ablaufplan Echtzeitsysteme	Stapelbetrieb, schwache & feste Echtzeitsysteme
Asymetrisch	Symmetrisch
	Identische Prozessoren ↳ Lastausgleich

Echtzeitsysteme

- Einhaltung vorgegebener Termine
- Rechtzeitigkeit anstelle von Geschwindigkeit
- Vorhersagbarkeit
- Überwachung/Interaktion mit physikal. Welt

Weiche Vorgaben (soft):
- Ergebnis weiterhin nutzbar
- Terminverletzung tolerierbar
- Transparent f. Anwendung

Feste Vorgaben (firm):
- Terminverletzung tolerierbar
- Abbruch der Berechnungen (durch BS)
  ↳ Ergebnis wertlos + Start der nächsten Berechnung (durch BS)
- Transparent für Anwendung

Harte Vorgaben (hard):
- Terminverletzung nicht tolerierbar
- BS löst Ausnahmesituation aus
- Ausnahmebehandlung führt System in sicheren Zustand
- Intransparent für Anwendung

Kriterien f. Prozesseinplanung

Benutzerorientiert: Vom Nutzer wahrgenommene Verhalten
- Mgl. kurze Antwortzeit (bei Sysfalls)
- " Durchlaufzeit (Start bis Ende)
- Termineinhaltung
- Vorhersagbarkeit

Systemorientiert: Mgl. effektive Auslastung von Betriebsmitteln
- Mgl. hoher Durchsatz an vollendeten Prozessen
- Gleichmäßige Prozessauslastung
- Gerechtigkeit
- Dringlichkeit
- Lastausgleich

Prioritäten nach Betriebsart

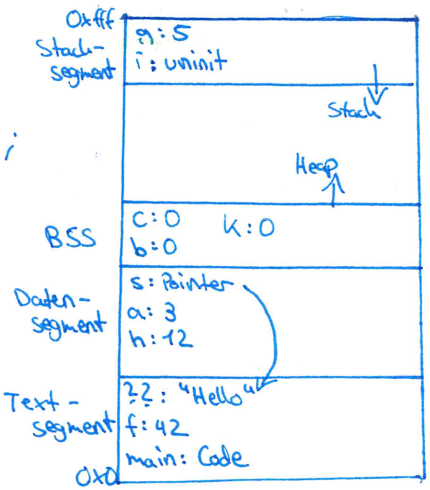
Allgemein: Gerechtigkeit, Lastausgleich
Stapelbetrieb: Durchsatz, Durchlaufzeit, Prozessorauslastung
Dialogbetrieb: Antwortzeit
Echtzeitbetrieb: Dringlichkeit, Termineinhaltung, Vorhersagbarkeit
Konflikt: Gerechtigkeit, Lastausgleich

Round Robin: FCFS mit Zeitscheibenverdrängung + period. Umplanung
VRR: Bevorzugung kürzerer Prozesse
Shortest Process Next: Verhungern möglich

```
Static int a = 3;
static int b;
static int c = 0;
const int f = 42;
const char *s = "Hello";
int main(void) {
    int g = 5;
    static int h = 12;
    int i;
    static int k;
}
```

Stack-Frame:

Alles was beim Fkt.-Aufruf und durch die Fkt. selbst auf den Stack gelegt wird.
Beim Rücksprung wird dieser Stack-Frame entfernt.



Adressraumschutz

→ Bei einem Prgr.

Abgrenzung/Abteilung: Ein Schutzgatterregister oberhalb erlaubten Bereich
Einfriedung/Eingrenzung: 2 Begrenzungsregister f. gültiges Intervall
↳ Bei beiden ext. Fragmentierung
Segmentierung: Basis- & Längenregister (↯ Pro Prgr. nur 1 Segment)

Fragmentierung:

Interne: Alle Blöcke gl. groß
Innerhalb der Blöcke ist der Speicher nur teilw. befüllt
↳ Kein Zugriffsfehler bei Zugriff auf ungenutzten Bereich

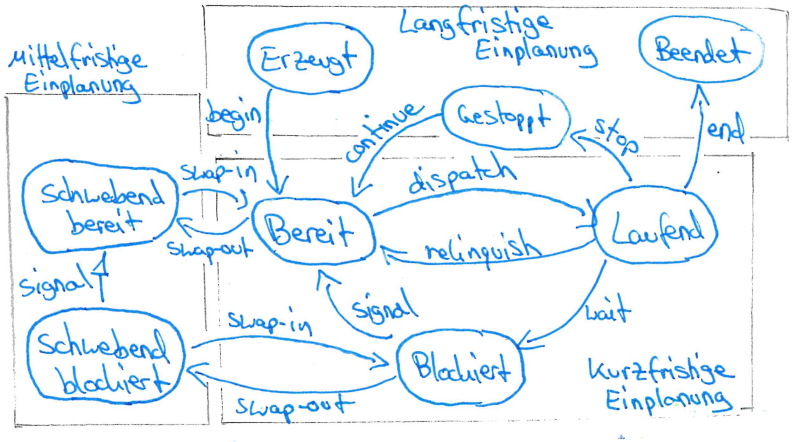
Externe: Blöcke unterschiedl. groß
Zwischen den Blöcken ist Speicher ungenutzt

Ersetzungsstrategien

FIFO, LRU: Am längsten
LFU: Am seltensten
2nd Chance: FIFO + period. use Bit
3rd Chance: ↳ + dirty-Bit bei mit 4. Zugriff Wiederholen

Page Fault

1. MMU erkennt nicht-gesetztes present bit → Trap
2. Wenn Seite ungültig: SIGSEGV (beendet)
3. Wenn Seite gültig: Seite wird angefordert und eingelagert (blockiert)
   - Falls in Freiseitenpuffer, nicht vom Hintergrundspeicher geladen werden
   - Falls keine freien Rahmen im Arbeitsspeicher: Ungenutzte Seite auslagern (nach Seitenersetzungsstrategie)
   - Seite in freien Rahmen einlagern und als present markieren (bereit)
4. Zugriff Wiederholen



Server-Socket

SIGPIPE ignorieren

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

```
int sock = socket(AF_INET6, SOCK_STREAM,0);
if(sock ==-1) die("socket");
struct sockaddr_in6 name = {
    .sin6_family = AF_INET6,
    .sin6_port = htons(port),
    .sin6_addr= in6addr_any, };
if(bind(sock, (struct sockaddr *)name,
    sizeof(name)) == -1) die("bind");
if(listen(sock, SOMAXCONN) == -1)
    die("listen");
while(1){
    int client = accept(sock, NULL, NULL);
    if(client == -1){ perror("accept");
        continue;}
    ...
}
```

fgets:

```
char buf[n+1];
while(fgets(buf, sizeof(buf), stream)!=NULL){
 size_t len = strlen(buf);
 if(len > sizeof(buf)-2 && buf[len-1] != '\n'){
    int c;
    while((c=fgetc(stream)) != EOF){
       if(c == '\n') break;
    }
    if(ferror(stream)) die("fgetc");
    continue;
 }
 if(buf[len-1] == '\n')
    buf[len-1] = '\0';
 ....
}
if(ferror(stream)) die("fgets");
```

Funktionen

realloc(a, sizeof(a)+10*sizeof(int));

strcpy(dest, src); strcat(dest, src);

```
char *saveptr;
strtok_r(str, "/", &saveptr);
```

fgetc(rx) != EOF; fgets(buf,size,rx) != NULL

fputc(rx) != EOF; fputs => fprintf ∇

Signale

Signal-Handler

```
struct sigaction a = {
    .sa_handler = &signalHandler,
    .sa_flags = SA_RESTART,};
sigemptyset(&a.sa_mask);
sigaction(SIGUSR1, &a, NULL);
ooo
static void signalHandler(int s){
    int tmperrno = errno;
    ...
    errno = tmperrno;
}
```

Signal blockieren

```
static volatile e =0;
...
e=0;
sigset_t oldmask, mask;
sigemptyset(&mask);
sigaddset(&mask, SIGUSR1);
sigprocmask(SIG_BLOCK, &mask,
    &oldmask);
```

Auf Signal warten

```
while(e==0)
    sigsuspend(&oldmask);
```

Signal deblockieren

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

strtol

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

Makefile

```
.PHONY: all clean
all: clash
clean:
    rm -f clash clash.o
clash: clash.o
    gcc -o clash clash.o
clash.o: clash.c clash.h
    gcc -c clash.c
```

Waitpid

```
int status;
pid_t child = waitpid(-1, &status, WNOHANG);
if(child == -1) perror("waitpid");
else if(child ==0) //no childs terminated
else{...
  if(!WIFEXITED(status)){
     fprintf(stderr, "%d\n",
        WEXITSTATUS(status));
  }
}
```

Ganzes Directory

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

Mutex (Semaphore)

```
static volatile int i = 0;
static pthread_mutex_t m;
static pthread_cond_t c;

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

void VC(){
   pthread_mutex_lock(&m);
   i++;
   pthread_cond_signal(&c);
   pthread_mutex_unlock(&m);
}
```

Semaphor

Koordinationsmittel zur Synchronisation von Betriebsmitteln. Atomarer Zähler:
PC(): Blockiert bei ≤0 (vorher dekr.)
VC(): Hebt Blockierung evtl. auf & erhöht

dup2

```
int fd = creat("path", 0);
if(fd == -1) die("creat");
if(fflush(stdout) == EOF)
   die("fflush");
if(dup2(fd, STDOUT_FILENO)
     == -1) die("dup2");
if(close(fd) == -1) perror("close");
```

Int-Overflow

```
if(x+y >
   INT_MAX)..
```

Atomics

```
atomic_int a = ATOMIC_VAR_INIT(10);
int d = atomic_load(&a);
atomic_store(&a, 8);
atomic_fetch_add(&a, 1); //a++;

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

Funktionen (2):

```
pthread_t p;
errno = pthread_create(&p, NULL, f, args);
pthread_detach(pthread_self());
errno = pthread_join(p, NULL);
errno = pthread_mutex_init(&m, NULL);
errno = pthread_cond_init(&c, NULL);
```

FILE *fopen("path", "r"); fclose(FILE *) //-1
FILE *fdopen(fd, "r"); close(fd) //-1
open("path",0) //-1; fileno(FILE *)

Journaling-FS:

Alle Änderungen an Daten & Metadaten werden vor Änderung als Transaktion (mit Infos zu Undo & Redo) protokolliert. Erfolgr. Ausführung wird auch protokolliert. Nach Absturz kann System wieder in einen konsistenten Zustand gebracht werden. Nach Erreichen best. Checkpoints wird Log-File gelöscht.

Platzierungsstrategien

Nach Größe sortiert:

Worst-	Best-Fit
schnelle	langsame Suche
größer	kleiner Verschnitt

Nach Adresse sortiert:

First-	Next-Fit
schnelle Suche	Round Robin
größer Verschnitt	v. First-Fit

Checkliste

- fflush stdout
- perror bei printf<0
- Argc immer prüfen!
- Alles static!

Inode:

Eigentümer, Dateigr., T4p, Letzter Zugriff
Page: Rechte, Present

RAID:

RAID 0: Verteilt
1: Kopien
4: XOR
5: XOR verteilt

Trap: Synchron, vorhersagbar
Interne Ursache
Beendigung / Wiederaufnahme
Interrupt: Asynchron, unvorhersagbar
Ext. Ursache
Wiederaufnahme