Speichersegmente

Text-/codesegment
- Programmcode
- statisch definierte Strings: "Hello"
- initialisierte const Variablen (read-only-Dinge)

Datensegment:
- initialisierte static variablen + globale variablen außerhalb von Funktionen

BSS-Segment:
- uninitialisierte nicht-lokale Variablen

Heap = malloc

Stack
- lokale Variablen einer Funktion + Argumente

Trap: interne Ursache
- synchron, vorhersagbar, reproduzierbar
- Beendigungs- oder Wiederaufnahmemodell (syscalls, pagefault (lokal) unbek. Gerät, Adressraum verletzt)

Interrupt: externe Ursache
- asynchron, meist von außerhalb der Maschine
- Nur Wiederaufnahmem. muss selten effektiv sein bzw.

Merkmale Prozessverw.:
Nutzer: Verbesserung von Vorhersagbarkeit, Durchlauf- und Antwortzeiten, Unabhängigkeit
System: gleichmäßige Auslastung, Gerechtigkeit, Ausgewogenheit, Vermeidung von Leerläufen

Ersetzungsstrategie
- FIFO: am längsten unbenutzt
- LFU: am seltensten benutzt
- 2nd-chance: FIFO + 1 Bit (wird auf 0 gesetzt, wenn unbenutzt → 0 zweimal sonst 1)
- 3Chance: FIFO + 2 Bits (0,0) ungelesen, (0,1) gelesen, (1,0) geschrieben

Wichtigkeit

Seitenfehler:
1 Seite nicht gefunden
2 MMU → Trap
3 Wenn Seite ungültig ist: Prozess beenden
4 Sonst erneut anfordern (ausgelagert)

Verklemmung:
irreversible Blockierung ohne Fortschritt

Bedingungen:
- Kein Ressourcenentzug
- gegenseitiger Ausschluss
+ zirkuläres Warten

Vermeidung:
Bankiers alg.: Überprüfung auf unsicheren Zustand vor Ressourcenzuteilung
- atomares Anfordern
- feste Reihenfolge
- Virtualisierung → Entzug
- nicht-blockierende Synchro.

Prozesseinplanung:
Kooperativ:
Kein Entzug, FCFS, am besten für Stapelbetrieb, - Monop.

Präemptiv:
CPU-Entzug ⇒ Keine Monopolisierung
Zeitscheiben mit Verdrängung (RR)

Deterministisch:
Prozesszeiten vor Laufzeit bekannt
→ Daten Becken für Echtzeit-OS

Probabilistisch:
shortest process first, da Zeiten unbekannt

statisch:
Vorgeplante Einplanung

dynamisch:
Einplanung zur Laufzeit

Symmetrisch:
Prozessoren sind gleich

assymetrisch: beachten, dass Prozessoren ungleich sind (Cons)

Nicht-Blockierende Synchronisation
Optimistischer Ansatz:
Wechselseitiger Ausschluss ist unhäufig, ⇒ CAS bis Erfolg
meistens direkt beim 1. mal
⇔ Pessimistischer Ansatz:
zu viel Konkurrenz ⇒ Synchronisation muss blockierend sein
+ umsetzen auf Anwendungsebene
+ Keine Systemaufrufe
+ geringe Wartezeit (nach OA)
+ Kein Ausschluss ⇒ Verklemmung
- Komplex
- Verhungern (Livelock)
- Hardware Unterstützung

Adressraumschutz:
Abgrenzung: Schutzgatterregister

Zonen: OS-zone und zone für Maschinenprogramme
OS-geschützt, MP nicht
restriktiv, interne Fragmentierung

Eingrenzung:
Ein Bereich im physikalischen Adressraum, worauf alle Speicherzugriffe beschränkt sind (von Begrenzungsregistern angelegt)

Segmentierung:
Basis- und Längenregister legen Segment/Scope für ein Programm (-) fest

Prozesse
Kernel-Thread:
- Scheduling durch OS
- kein eigener Adressraum
- Systemaufrufe blockieren andere Kernel-Threads nicht
- sind für Multiprozessoren geeignet

User-Thread:
- scheduelling durch Entwickler, nicht OS
- vom Kernel weder gesehen noch verwaltet
- System calls blockieren andere User Threads
- ungeeignet für Multiprozessoren
- unpriv. ligster Code

Prozesse $\subsetneq$ KernelThreads
+ eigener Adressraum

Zuteilungsverfahren:
Worstfit: erstes größtes passendes Loch, großer Verschnitt, hoher Aufwand
bestfitt: kleinstes passendes Loch, großer Aufwand, kleiner Verschnitt
} nach Lochgröße sortiert

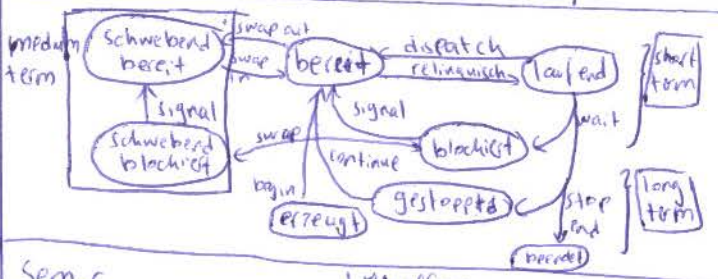
firstfit: nach Adressgröße sortiert, erstes passendes Loch, kleiner Aufwand, großer Verschnitt
nextfit: firstfit + (RR) Suche ab dem letzten zugeteilten Block

Buddy-Verfahren
1. runde aufgeforderten Speicher auf die nächste 2^n Zahl auf
2. Suche nach einem Block der Größe 2^n
   2.1 (nicht gefunden) erhöhe n um 1 und gehe zu 2.
   2.2 Block zuteilen
3. den 2^n Block allokieren
- zwei Blöcke gleicher Größe werden zurückverschmolzen zu einem bei Freigabe
- Aufrunden ⇒ interne, aber keine externe F.

Inode
Inode Nummer := st_ino | Größe := st_size
Referenzzähler := st_nLink

Inode Parameter → Inhalt

Inhalt
- Ordner: st_ino + (, + ... + jede Datei)
- Datei: ???
- Verknüpfung: verknüpfte Datei (ihr Pfad)



RAID:
0. Verteilen, schnell lesen, nicht sicher
1. Spiegeln, Schutz vor Verlust
Verlust: 0 - eine, 1 - alle bis auf eine
4. 1-Parität, n-1-Data
Aufwand auf eine Platte
5. Parität und Daten alt. allem

Sem C
```
Static volatile int i = 0;
Static Pthread_mutex_t m;
Static Pthread_cond_t c;
Void P(){
  Pthread_mutex_lock(&m);
  while(i<=0){
    Pthread_cond_wait(&c,&m);
  }
  i--;
  Pthread_mutex_unlock(&m);
}
Void V(){
Pthread_mutex_lock(&m);
i++;
Pthread_cond_signal(&c);
Pthread_mutex_unlock(&m);
}
```
V() freigeben, inkrementieren
P()

Sbuffer C
```
Struct BNDBUF {
  int read, write;
  size_t size;
  SEM *sr, sw;
  int values[];
};

BNDBUF *bbCreate(size_t size){initialisierung} //error psb.
Void bbDestroy(BNDBUF *bb);
Void bbPut(BNDBUF *bb, int i);
int bbGet(BNDBUF *bb);
```

Server

```
//SIGPIPE Ignorieren
int listensock = socket(AF_INET6, SOCK_STREAM, 0);
if(listensock == -1) die("socket");
struct sockaddr_in6 name = {
        .sin6_family = AF_INET6,
        .sin6_port = htons(PORT)
        .sin6_addr = in6addr_any, };
if(bind(listensock, (struct sockaddr*)name,
    sizeof(name)) == -1) die("bind");
if(listen(sock, SOMAXCONN) == -1) die("listen");
While(1){ //accept clients
 int clientsock = accept(listensock, NULL, NULL);
 if(clientsock) == -1) { perror("accept");
                         continue; }
}
```

Signalbehandlung

```
struct sigaction a = {
        .sa_handler = &sighandler
        .sa_flags = SA_RESTART, };
sigemptyset(&a, sa_mask);
static void sighandler(int sig); — sigaction(unten)
int temperrno = errno
errno = temperrno;
}
```

Ignorieren:

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

Blockieren

```
sigset old, mask;
sigemptyset(&mask, SIGANY);
sigprocmask(SIG_BLOCK, &mask, &old); //-1
While(!cond){ sigsuspend(&old);
sigprocmask(SIG_UNBLOCK, &old, NULL); //-1
```

Open FILE*

```
FILE*f fopen(path, mode);
FILE*f = fdopen(fd, mode);
mode ∈ {"r","w","a"}
if(f == NULL) die("fopen");
if((f)close(FILE*/fd) == -1) die("close");
```

fgets

```
char buf[SIZE];                      FILE*
while(fgets(buf, sizeof(buf), fp){   printf
        != NULL){ ... }
if(ferror(fp)) die("fgets");
```

fgetc

```
int c;
while((c = fgetc(fp)) != EOF){..}
if(c != EOF && ferror(fp)) die("fgetc");
```

fork, waitpid

```
pid_t pid = fork();
if(pid == -1) die("fork");
if(pid == 0) { //child prozess}
if(pid > 0) {//parent }
//oder in //parent| int status;
pid_t child = waitpid(pid, &status, WNOHANG);
//WNOHANG in while(waitpid ...), sonst 0
//pid kann -1 sein (wait for any child
if(child == -1) die("waitpid");
else if(child == 0) // keine child p. terminiert
else{//status auf WIFEXITED prüfen}
 if(!WIFEXITED(status))//errors
```

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

readdir nicht Thrd-safe -> scandir

Parse_Int (strtol)

```
errno = 0;
char * endptr;   (argument*)  (Dec)
int x = strtol(str, &endptr, 10);
if(errno != 0) die("strtol");
if(endptr == str || endptr != '\0'){}
//invalid argument in str
```

Pthread_stuff

```
pthread_t p;
pthread_create(&p, NULL, function, NULL)
// if != 0, die
pthread_join(&p, NULL); //wartet, != 0
pthread_detach(&p); //markieren als
detached, Ressourcen werden automatisch
gesäubert
```

ScanDir

```
struct dirent ** namelist;   (some filter)
int n = scandir(path, &namelist, alphasort);
while(n--){ ... free(namelist[n]); }
free(namelist);
```

Socket Verbindung behandeln

```
While(1){
 int sock = bbGet();
 FILE*rx = fdopen(sock, "r");
 if(rx == NULL) { close(sock);
                  continue; }
 int sock2 = dup(sock);
 if(sock2 < 0) {close(sock); ...}
 FILE* tx = fdopen(sock2, "w");
 if(tx == NULL) {fclose(rx);
                 close(s2); Cont;
 handleDialog(rx, tx);
 fclose(rx);
 fclose(tx);
```

Dup2 - Std-Stream umleiten

Stdin

```
int fd = open(file, O_RDONLY) //=-1 die
if(dup2(fd, STDIN_FILENO)) == -1 die
if(close(fd) == -1) die
stdout S_IWUSR ... rw-r--r--
int fd = creat(file, mode);
dup2(fd, STDOUT_FILENO);
```

CAS

```
atomic_int a = ATOMIC_VAR_INIT(10);
int d = atomic_load(&a);
atomic_store(&a, b);
atomic_fetch_add(&a, 1); //a++
int old, local;
do{
 old = atomic_load(&a);
 local = old + something;}
while(!atomic_compare_exchange_strong(
  &a, &old, local));
```

```
realloc(a, sizeof(a) * 2);
strcpy(dest, src); //same for strcat
strtok_r(str, delim, &saveptr);
```

Client:

```
char*host, *port;
struct addrinfo hints = {
   .ai_socktype = SOCKSTREAM,
   .ai_family = AF_UNSPEC
   .ai_flags = AI_ADDRCONFIG, };
struct addrinfo*head;
int err = getaddrinfo(URL, "80", &hints, &head
if(err != 0){
  if(err = EAI_System) perror("getaddrinfo")
}else{ fprintf(stderr, "getaddrinfo: %s"
                  gai_strerror(err)); }
int sock;
for(struct addrinfo curr = head; curr != NULL;
    curr = curr->ai_next){
 sock = socket(curr->ai_family,
               curr->ai_socktype,
               curr->ai_addrlen);
 if(sock == -1) continue;
 if(connect(sock, curr->ai_addr,
            curr->ai_addrlen) == 0)
   break; // verbunden
}
close(sock);
```