```c
static void die(char *s){
    perror(s); exit(1);
}

static int (*cmp)(const void *, const void *)
    = (int (*)(const void *, const void *))compare;
```

DATEIEN

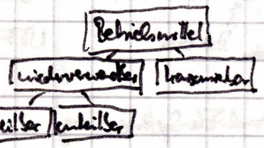
```c
char buf[100]; // 99 + '\0'
FILE *fh = fopen("path", "r");
if (fh == NULL){
    die("fopen");
}
while (fgets(buf, (int)100, fh) != NULL){
    printf("%s", buf);
}
if (ferror(fh)){
    die("fgets");
}
fclose(fh); // ggf. if (fclose(fh) != 0) {die("_");}
```

VERZEICHNISSE

```c
DIR *dir = opendir("path"); struct dirent *entry;
if (dir == NULL){
    die("opendir");
}
while (errno = 0, (entry = readdir(dir)) != NULL){
    if (strcmp(entry->d_name, ".") == 0 ||
        strcmp(entry->d_name, "..") == 0){
            continue;
    }
    char path[strlen("path") + strlen(entry->d_name) + 2];
    sprintf(path, "%s/%s", "path", entry->d_name);
    struct stat info;
    if (lstat(path, &info) != 0){
        continue; // ggf. perror(..);
    }
    if (!S_ISREG(info.st_mode) || !S_ISDIR(info.st_mode)){
        continue;
    }
    printf("%s: %ld Bytes\n", path, info.st_size);
}
if (errno != 0){
    die("readdir");
}
closedir(dir); // ggf. if (closedir(dir) != 0) { die(".."); }
```

THREADS

```c
pthread_t thread;                 (void *(*)(void *))   (void *)
int result = pthread_create(&thread, NULL, thread_start, argument);
if (result != 0){
    die("pthread_create");
}
result = pthread_detach(thread);
if (result != 0){
    die("..");
}
```



FORK

```c
pid_t pid = fork();
if (pid == 0){
    // child
} else if (pid == -1){
    die("fork");
}
int status;
if (waitpid(pid, &status, 0) == -1){
    die("waitpid");
}
if (WIFEXITED(status)){
    printf("Exit: %d\n", WEXITSTATUS(status));
} else if (WIFSIGNALED(status)){
    printf("Signal: %d\n", WTERMSIG(status));
} else {
    printf("Unknown\n");
}

int status;
while ((pid = waitpid(-1, &status, WNOHANG)) != 0){
    if (pid == -1){
        if (errno == ECHILD){
            break; // no children to collect
        }
        die("waitpid");
    }
}
```

CAS BSP:

```c
do {
    pos = bb->read_pointer;
    pos_new = (pos + 1) % bbsize;
    fd = bb->buffer[pos];
} while (!CAS(&bb->read_pointer,
              pos,
              pos_new));
```

SOCKETS CLIENT

```c
struct addrinfo gai_hints;
struct addrinfo *gai_result, *server;
int gai_return, server_socket;
[redacted] *server_fd;
memset(&gai_hints, 0, sizeof(gai_hints));
gai_hints.ai_family   = AF_UNSPEC;
gai_hints.ai_socktype = SOCK_STREAM;
gai_hints.protocol    = 0;
gai_hints.ai_flags    = AI_NUMERICSERV
                      | AI_ADDRCONFIG
                      | AI_V4MAPPED;
gai_return = getaddrinfo(host, port, &gai_hints, &gai_result);
if (gai_return != 0){
    die("..");
}
for (server = gai_result; server != NULL; server = server->ai_next){
    server_socket = socket(server->ai_family, server->ai_socktype, server->ai_protocol);
    if (server_socket == -1){
        continue; // ggf. perror("..");
    }
    if (connect(server_socket, server->ai_addr, server->ai_addrlen) != -1){
        break; // successful connected
    }
    if (close(server_socket) != 0){
        perror("..");
    }
}
freeaddrinfo(gai_result);
if (server == NULL){
    die("no server");
}
```

SOCKETS READ

```c
server_fd = [redacted] fdopen(server_socket, "a+");
if (server_fd == NULL){
    close(server_socket);
    die("fdopen");
}
int result = fscanf(server_fd, "%d", &value);
if (result == 0){
    fclose(server_fd); die("_"); // or no die!
} else if (result == EOF && !ferror(server_fd)){
    fclose(_); die("..");
} else if (ferror(server_fd)){
    fclose(..); die("..");
}
```

SOCKETS SERVER

```c
char *port_string = "4711";
int port = atoi(port_string); // check <=0, etc.
int s = socket(AF_INET6, SOCK_STREAM, 0);
if (s == -1){
    die("..");
}
struct sockaddr_in6 sin;
memset(&sin, 0, sizeof(sin));
sin.sin6_family = AF_INET6;
sin.sin6_addr   = in6addr_any;
sin.sin6_port   = htons(port);
if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) == -1){
    die("bind");
}
if ([redacted] listen(s, 5) == -1){
    die("..");
}
struct sockaddr_in from;
socklen_t from_length = sizeof(from);
for(;;) {
    int cin = accept(s, (struct sockaddr *)from, from_length);
    if (cin == -1){
        continue; // ggf. perror()
    }
    ...
}
close(s); // ggf. if (close(s) != 0){.. die(_);}
```

SIGNALS

```c
struct sigaction action;
action.sa_flags = SA_NOCLDSTOP | SA_RESTART;
action.sa_handler = (void (*)(int)) ghostbuster; // SIG_IGN
sigaction(SIGCHLD, &action, NULL);

sigset_t set;
sigemptyset(&set);
sigaddset(&set, SIGCHLD);
sigprocmask(SIG_BLOCK, &set, NULL); // SIG_UNBLOCK
errno!
```

FUNKTIONEN

```c
void *memcpy(void *dest, const void *src, size_t bytes);
char *strcpy(char *dest, const char *src);

int strcmp(const char *s1, const char *s2);
char *strstr(const char *haystack, const char *needle);
```

SEMAPHORE

```c
pthread_mutex_t mutex;        int v = 0;
pthread_cond_t cond;
pthread_mutex_init(&mutex, NULL); // errno check: != 0
pthread_cond_init(&cond, NULL);   //      "

P() {
    pthread_mutex_lock(&mutex);
    while (v == 0) {
        pthread_cond_wait(&cond, &mutex);
    }
    v--;
    pthread_mutex_unlock(&mutex);
}

V() {
    pthread_mutex_lock(&mutex);
    v++;
    pthread_cond_broadcast(&cond);
    pthread_mutex_unlock(&mutex);
}

pthread_cond_destroy(&cond);
pthread_mutex_destroy(&mutex);
```

FORTSCHRITTSGARANTIEN

Behinderungsfrei: in Isolation laufender Faden beendet seine Operation in begrenzter Anzahl von Schritten
Sperrfrei: garantiert systemweiten Fortschritt, Verhungern möglich
Wartefrei: kein Verhungern

MONITOR

Hansen: beidseitig blockierende Bedingungsvariablen, signal(): alle Prozesse befreien, Neuauswertung der Wartebedingung nötig
Mesa [redacted]: einseitige ~, signal(): einen oder alle befreien, Neuauswertung nötig, Gefahr der Prioritätsverletzung
Hoare: beidseitige, signal(): höchstens einen befreien, keine Neuauswertung nötig, Prioritätsverletzung

CAS

ABA-Problem: Lösung: Generationszähler, oder LL/SC

BETRIEBSMITTELVERWALTUNG

Konflikt: begrenzte Anzahl von Betriebsmitteln, unteilbar, gleichzeitiger Zugriff
Streit: Anforderung eines Betriebsmittels, das schon gelockt wurde

SEMAPHORE

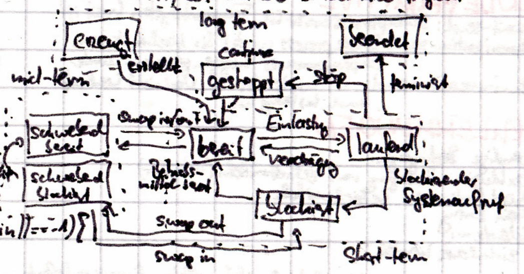
Binärer: gegenseitiger Ausschluss (Mutex), unteilbare Betriebsmittel, [0,1]
Zählender ~: konsumierbare Betriebsmittel, Limitierung von Ressourcen, [0,N]
— : Koordination/Signalisierung, benachrichtigt andere Threads (Erzeuger/Verbraucher)

PROGRAMM

Programm: Folge von Anweisungen
Prozess: Programm das sich in Ausführung befindet (abstrakt)
Prozessinkarnation: konkrete Ausführungsumgebung für ein Program.

PROZESSZUSTÄNDE

erzeugt: besitzt noch nicht alle nötigen Betriebsmittel
bereit: bereit zum Laufen
laufend: wird gerade ausgeführt
blockiert: wartet auf Ereignisse (I/O, Betriebsmittel, Nachricht)
beendet: terminiert, noch nicht alle Betriebsmittel freigeben



UNTERBRECHUNGEN

Trap: synchron, vorhersagbar, reproduzierbar, programmierbar (segfault, pagefault, illegale Anweisung)
Interrupt: asynchron, unvorhersagbar, nicht ~ (Tastendruck, DMA, E/A beendet, Seitenfehler bei globaler Ersetzungsstrategie)

Wiederaufnahmemodell: Fortsetzung: Trap kann, Interrupt muss
Beendigungsmodell: Trap kann, Interrupt darf nie

ARBEITSSPEICHER

physikalischer Adressraum: Hardware, hat Lücken, eindeutig
logischer Adressraum: Compiler, Binder, BS, alle Adressen gültig, mehrdeutig (je Prozess)
virtueller Adressraum: BS, abgebildeter logischer ~, mehrdeutig

Programm → logischer ~ (→ virtueller ~) → physikalischer ~

ADRESSRAUMSCHUTZ

Abteilung: Schutzgatter trennen BS und Programm, ggf. Schutzgatterregister, statisch vor Programmausführung, physikalischer ~
Eingrenzung: Begrenzungsregister pro Programm, Abbildung auf physikalischen ~, statisch zur Ladezeit, Hochwasserregister, verschiebender Lader, CPU prüft Adressen
Segmentierung: Basis/Längenregister bilden logische ~, bei Einlastung Abbildung auf Hardwareregister, dynamisch zur Laufzeit, Lader bestimmt physikalische Adresse, logischer ~, CPU/MMU besorgt Bindung zur Ausführungszeit

ECHTZEITSYSTEME

weich: verspätete Ergebnisse nutzlos, Terminverletzung tolerierbar
fest: " werden verworfen, " ", Ergebnis verworfen
hart: " Katastrophe, " nicht tolerierbar

EINPLANUNGSVERFAHREN

cooperative scheduling: Systemaufrufe nötig um Prozesse zu wechseln, Monopolisierung der CPU nötig
preemptive ~: Prozessen wird die CPU entzogen, CPU-Schutz
deterministische ~: CPU-Stoßlängen und Termine bekannt, genaue Vorhersage der CPU-Auslastung möglich
probabilistische ~: nur Abschätzung möglich
offline ~: vor der Programmausführung, strikte Echtzeitsysteme
online ~: während der "

FCFS first come first served: kooperativ, FIFO, Konvoieffekt (kurze CPU-Stöße nach langen)
RR round robin: Zeitscheibe pro Prozess (periodische Unterbrechungen), CPU-Schutz. Problem: kurze CPU-Stöße folgen langen, E/A intensive Prozesse benachteiligt
VRR virtual RR: Prozesse werden nach E/A-Stoß bevorzugt eingeplant, bessere Verteilung der CPU-Auslastung
SPN shortest process next: kooperativ, a priori Wissen über Laufzeiten notwendig, Verhungern möglich
SRTF shortest remaining time first: ereignisbedingte Umplanung (E/A), Verdrängung
HRRN highest response ratio next: Einplanung nach erwarteter Bedien- und aktueller Wartezeit, beugt Verhungern vor
MLQ multilevel queue: Prozesse nach Typ eingeplant mit lokalen Strategien, statisch/dynamisch, globale Einplanung zwischen den Listen
FB feedback: kurze/interaktive Prozesse bevorzugt

kooperativ: FCFS, SPN   probabilistisch: SPN, SRTF, HRRN
verdrängend: RR, VRR   mehrstufig: MLQ, FB (MLFQ)

PROZESSKONTROLLBLOCK

- Speicher-, -Adressraumbelegung
- Dateideskriptoren, -köpfe (inodes)
- Programmdatei
- Laufzeitkontext
- Scheduling-Informationen
- Ereignisse
- Benutzerzuordnung, -rechte

KRITISCHE ABSCHNITTE

keine Verdrängung erlaubt
Verdrängung wieder möglich

- Interrupts abschalten: cli/sti
- NPCS (non-preemptive-critical-section) enter/leave, Verdrängungssperre
- Semaphore: P/V
- Bedingungsvariable: im KA warten, diese im Blockieren
  Zähl-, Sperr-, Bedingungsvariable
- Schlossvariable: busy-lock (mit Minimierung von TAS)

await: lässt Prozess auf Bedingungsvariable warten und gibt KA frei, bei Signal erneut Betreten des KA
cause: löst mit der ~ verknüpftes Ereignis aus, deblockiert wartende Prozess

GEWICHTSKLASSE PROZESSE

schwergewichtiger Prozess: Prozessinkarnation und Benutzeradressraum bilden eine Einheit
Prozesswechsel: zwei Adressraumwechsel: $AR_x \rightarrow BS \rightarrow AR_y$
„klassische" UNIX-Prozess

leichtgewichtiger Prozess: Prozessinkarnation und Adressraum sind entkoppelt
Prozesswechsel: ein Adressraumwechsel: $AR_x \rightarrow BS \rightarrow AR_x$
Kernfaden

federgewichtige Prozesse: Prozessinkarnationen und Adressraum bilden eine Einheit
Prozesswechsel: keine Adressraumwechsel: $AR_x \rightarrow AR_x$
Benutzerfaden

SENDEPRIMITIVE

no-wait-send: Sendeprozess wartet bis Transportsystem bereit ist
synchronization-send: Sendeprozess wartet bis Nachricht vom Empfangsprozess entgegengenommen wurde
remote-invocation-send: ~ wartet bis ~ Nachricht bearbeitet und bestätigt hat

VERKLEMMUNGEN

notwendige Bedingungen:
- exklusive Belegung von Betriebsmitteln (unteilbar)
- Nachfordern von Betriebsmitteln
- kein Entzug

hinreichende ~:
- zirkuläres Warten

Betriebsmittelgraph: zeigt pro Betriebsmittel welcher Prozess es belegt
Wartegraph: zeigt Betriebsmittel auf die ein Prozess wartet (geschlossener Kreis zeigt Deadlock)

~ VORBEUGUNG

verhindern Verklemmungen zur Entwurfs-, Implementierungszeit
- nicht-blockierende Verfahren
- Betriebsmittelanforderungen unteilbar machen
- Betriebsmittelentzug durch Virtualisierung erlauben
- lineare / totale Ordnung der Betriebsmittel

~ VERMEIDUNG

fortlaufende Bedarfsanalyse schließt zirkuläres Warten aus

~ ERKENNUNG

Wartegraph erstellen und Zyklen suchen ($O(n^2)$), Prozess terminieren

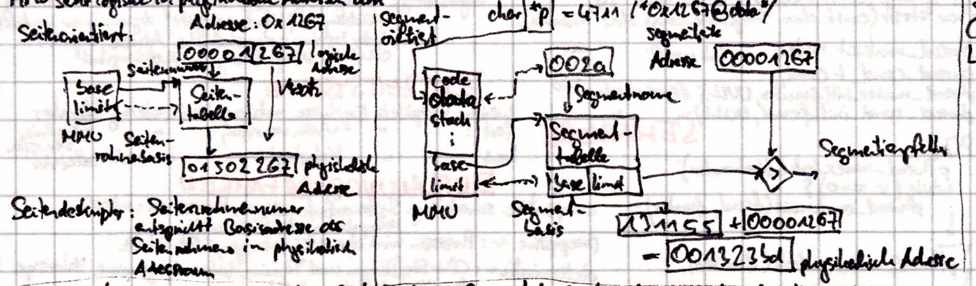
~ SFREIHEIT

sicherer Zustand: es existiert Folge der Verarbeitung vorhandener Prozesse in der alle Betriebsmittelanforderungen erfüllbar sind
unsicherer ~: Folge existiert nicht, Erkennung durch Betriebsmittel-Belegungsgraph ($O(n^2)$), Bankiersalgorithmus

ARBEITSSPEICHER II

Seitennummerierung (paging): Fragment: Vielfaches von Seitenrahmen, interne Fragmentierung
Adresse: Tupel (p,o): p Seitennummer, o Versatz in Seite
Segmentierung (segmented): Fragment: Vielfaches von Bytes, externe Fragmentierung
Adresse: Paar (S,A): S Segmentnummer/name, A Adresse relativ zu S

MMU setzt logische in physikalische Adressen um



Seitendeskriptor: Seitenrahmennummer entspricht Basisadresse des Seitenrahmens im physikalischen Adressraum

kombiniert: zuerst Segment, dann Seitenorientiert, Segmenttabellen base/limit zeigt auf Seitentabelle

Segment/Seitendeskriptor:
- Basisadresse im physikalischen Adressraum
- Zugriffsrechte (lesen/schreiben/ausführen)

Segmentdeskriptor:
+ Segmentlänge
+ Expansionsrichtung (Halde / Stapel)

TLB = translation lookaside buffer

- present bit (0 = ausgelagert)
Basisadresse gilt für Hintergrundspeicher

FREISPEICHER

- Bitkarte: für seitennummerierte Adressräume
- Freispeicherliste: für segmentierte "
hat Anfangsadresse und Länge
  - Liste und Löcher getrennt (belegt Betriebsmittel), Adresse keine Adressraumumschaltung
  - Liste und Löcher vereint (keine "), eine "
  - fore, rear, next, size, base

best-fit: Löcher aufsteigender Größe (Verschnitt minimieren)
worst-fit: Löcher absteigender Größe (Suchaufwand minimieren)
buddy: Kompromiss

first-fit: Löcher der Adresse nach sortiert (Verwaltungsaufwand minimieren)
next-fit: ", beginnt bei letztem Loch (Suchaufwand minimieren)

Verschmelzung / Kompaktifizierung zur Reduzierung der externen Fragmentierung
Verschiebung

VIRTUELLER SPEICHER

Seitenüberlagerung (Paging): Adressraumteile fester Größe werden auf Seitenrahmen des Hauptspeichers abgebildet; Problem: interne Fragmentierung
Segmentierung: Adressraumteile variabler Größe werden abgebildet; Problem: externe Fragmentierung
Seitennummerierte Segmentierung: Kombination, Segment in Seiten untergliedern

Einlagerung: on-demand: present-bit
Vorausladen: Heuristiken

Teilemulation des Maschinenbefehls durch das BS

Verdrängung: OPT (optimales Verfahren) Seite verdrängen die am längsten nicht mehr verwendet werden wird

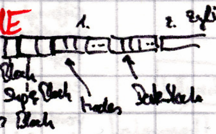
FIFO (first-in, first-out): zuerst eingelagerte Fragment
LFU (least frequently used): selten genutzte Fragment
LRU (least recently used): das kürzlich am wenigsten genutzte Fragment
Zählverfahren: LFU, MFU
Referenzzähler pro Seite, wird mit jeder Referenz erhöht, mittelmäßige Approximation OPT
Zeitverfahren: LRU, Zeitstempelwert / Stapel base ältesten Seite | Rückwärtsabstand zurückliegende Seitenreferenz: gute ~
LRU Aging: Referenzbit (1 = Zugriff), Speicherung in Schieberegister (>>)
am wenigsten genutzte Seite sind 0
LRU second chance: Referenzbit 1: auf 0 setzen, zweite Chance | keine Unterscheidung lesend/schreibende Zugriffe
0: Ersetzungskandidat
~enhanced second chance: modify/dirty bit mit nutzen;
(R|D): (0,0) beste Wahl
(0,1) keine schlechte Wahl (nur beschrieben)
(1,0) keine gute Wahl (gelesen)
(1,1) schlechteste Wahl

Freiseitenpuffer: nutzt Vorlauf, über Schwellwerte gesteuert
low: Seite ggf. auslagern
high: Seite ggf. einlagern (Vorausladen)
Freiseiten können in Zwischenspeicher bevor sie geswapped werden

Seitenüberlagerung: lokal: nur Seiten des Prozessraums betroffen (Trap)
global: alle Seiten betroffen (Unterbrechung) ← Seitenfehler, Seitenfehler / Referenzbits

Thrashing verhindern: Arbeitsmenge bestimmen (ähnlich zu LRU) und diese vorladen / aktiv lassen (komplett), Prozesse die Arbeitsmenge aussperren / stoppen

DATEISYSTEME



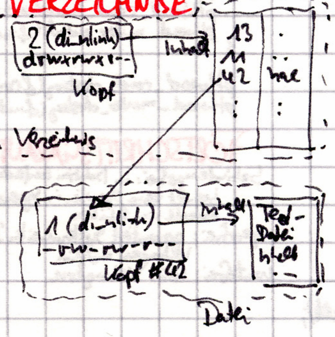
EXT2 analog, Zylinder ⇒ Block

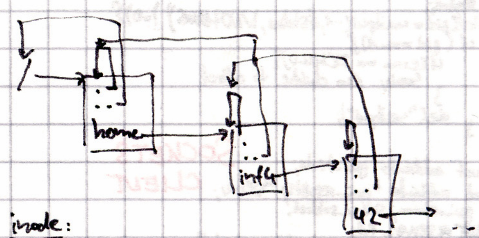
Master-File-Table-Eintrag: Länge, Standard-Attribute, Zeitstempel, Hardlink #, Sequenznummer

NTFS: Dateiname, Zugriffsrechte, Daten, ggf. weitere Streams: Extents (Erweiterung), Attributliste, Streams mit beliebigen Daten

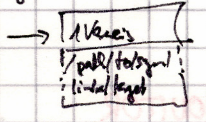
Journal-File-Systeme: Log
Copy-on-Write-FS
Log-Structured-FS: schreibt immer am Ende, gute Schreibeffizienz

VERZEICHNISE





Symlink



inode:
- uid, gid
- Typ: reguläre / spezielle Datei
- Rechte: lesen, schreiben, ausführen
- Zeitstempel (letzter Zugriff / Änderung)
- Anzahl der Verweise (hard links)
- Größe (bytes)
- Adresse(n) der Datei auf der Festplatte

inode-nummer:
Index in einer Tabelle von Dateiköpfen (inode number)