GEWICHTSKLASSE PROZESSE

Prozesskontrollblock / eigene Ressourcen

schwergewichtiger Prozess: Prozessinkarnation und Betriebsadressraum bilden Einheit
(klassischer UNIX-Prozess) Prozesswechsel: 2 Adr.raumwechsel $AR_x \to BS\ AR_y$
leichtgewichtiger Prozess: Prozessinkarnation und Adr. Raum sind entkoppelt
(Kernfaden) Prozesswechsel: 1 Adr.Raumwechsel $AR_x \to BS \to AR_x$
federgewichtiger Prozess: Prozessinkarnationen und Adressraum bilden Einheit
(User faden) Prozesswechsel: 0 Adr.Raumwechsel $AR_x \to AR_x$

VERKLEMMUNGEN

Notwendige Bedingungen:
① exkl. Belegung von BM (unteilbare BM) → mutual exclusion
② Nachfordern von BM → hold and wait
③ Kein Entzug von BM (BM nicht rückforderbar → no preemption
Hinreichende Bedingungen:
④ zirkuläres Warten (Kette, speisende Phil.) → circular wait
DEADLOCK: Stillstand von P bei passivem Warten
LIVELOCK: aktivem Warten
↳ busy waiting (P hat CPU), lazy waiting (P hat CPU nicht)
Konsumierbare BM: unbegrenzte Anzahl → einseitige Sync.
Reusable BM: begrenzte Anzahl → mehrseitige Sync.
BM Graph: zeigt pro BM welcher P es belegt
Warte Graph: zeigt BM auf die ein P wartet
Überlappender Zugriff aus gemeinsame unteilbare BM

VORBEUGUNG

verhindert Verklemmungen zur Entwurfs-/Implementierzeit
① nicht-blockierende Verfahren
② BM anforderungen unteilbar machen } indirekt gegen 1-3
③ BM entzug durch Virtualisierung ermöglichen
④ lineare / totale Ordnung von BM-Klassen einführen } direkt gg. 4

VERMEIDUNG

gegen „unsichere Zustände"
fortlaufende Bedarfsanalyse schließt zirkuläres Warten aus

ERKENNUNG

Wartegraph erstellen → Zyklen suchen ($O(n^2)$), P terminieren

VKSFREIHEIT

sicherer Zustand: ∃ Folge der Verarbeitung vorhandener P, in der alle BM-Anforderungen erfüllbar sind
unsicherer Zustand: ∄ Folge; Erkennung durch BM-Belegungsgraph, Bankiersalgorithmus

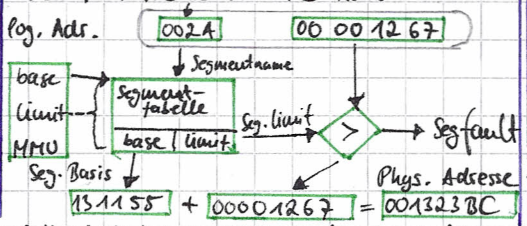
ARBEITSSPEICHER II

Seitennummerierung (pages): Fragment: Vielfaches von Seitenrahmen, interne Fragmentierung; Adresse: Tupel(p,o), p: Seitennr., o: offset
Segmentierung: Fragment: Vielfaches von Bytes, externe Fragmentierung
Adresse: Paar (S, A), S: Segmentnr. / name, A: Adr. rel. zu S

Seitenorientiert. Adresse: 0x1267
char *p = 4711

Seitendeskriptor: Seitenrahmennr. entspricht Basisadr. des Seitenrahmens im physikalischen Adressraum

Segmentorientiert;
char *p = 4711 @ 42 0x1267 @ 0x2a



Kombiniert: erst Segment, dann Seitenorientiert, Segmenttabellen base / limit zeigt auf Seitentabelle

Segment / Seitendeskriptor:
- Basisadresse im phys. Adressraum
- Zugriffsrechte (r, w, x)
- present bit (0 = ausgelagert)
Basisadresse gilt für Hintergrundspeicher

Segmentdeskriptor
- Segmentlänge
- Expansionsrichtung (Halde / Stapel)

TLB: Translation Lookaside Buffer

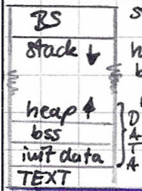
FREISPEICHERVERWALTUNG

jedes Bit steht für Speicherbereich fester Größe

- Bitkarte für seitennummerierte Adressräume
- Freispeicherliste für segmentierte " " ← Verschmelzung aufwändig
hat Anfangsadresse und Länge
① Liste und Löcher (adressmäßig) getrennt
sie belegen BM, entries sind Löcherdeskriptoren
② Liste und Löcher (adressräumlich) vereint
belegen keine BM, entries sind Löcher

zu ① fore, aor, next, size, base pro entry
best-fit: Löcher aufsteigender Größe (Verschnitt minimieren)
worst fit: Löcher absteigender Größe (Suchaufwand minimieren)
buddy: Kompromiss
first-fit: Löcher der Adr. nach sortiert ↑ (Verwaltungsaufwand ↓)
next-fit: reihum, beginnt beim letzten Loch (Suchaufwand ↓)
Verschmelzung / Kompaktifizierung (Verdichtung) zur Reduzierung der externen Fragmentierung

SPEICHERORGANISATION PROZESS



stack: lokale Variablen, Funktionsparameter, Speicher für Registerinhalte; dyn. erweiterbar
heap: dyn. erweiterbar bss → malloc
bss: nicht init. globale und static Variablen (mit 0 vorinitialisiert)
init: initialisierte globale und static Variablen
text: Programmcode

PROZESSE II

schwergewicht: Adr. Raum, Ressourcen, Prozesskontrollblock verwaltet vom Kern
leichtgew.: kein eigener Adr. Raum, gemeinsame Resourcen, vom BS-Kern verwaltet, eigener Threadkontrollblock
syscall
federgew.: Auf User-Lvl → BS kennt ihn nicht
Bibliothek

SEITENDESKRIPTOR

Seitenadresse (phys. Adr. der Seite im Hauptspeicher
Present-bit (ob Seite im Hauptspeicher)
Schutzbits (Rechte)
Referenced-bit (Ob auf Seite zugegriffen wurde
Dirty-bit (wenn auf Seite geschrieben wurde
Cache-Disable Bit et. al.

```
int main(int argc, char *argv[]){
return 0; }
```

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, Segmente 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 eingelagertes Fragment
LFU (least frequently used): selten genutztes Fragment (Var: MFU)
LRU (least recently used): das am längsten nicht genutzte Fragment
Zählverfahren: LFU, MFU; Referenzzähler pro Seite, wird mit jeder Referenz erhöht → mittelmäßige approx. von OPT
Zeitverfahren: LRU Zeitstempelwert / Stapelboden älteste Seite / Rückwärtsabstand zurückliegender Seitenreferenzierungen → gute approx.
LRU Aging: Referenzbit (1 = Zugriff), Speicherung in Schieberegister (>>)
am wenigsten genutzte Seiten sind 0
LRU 2nd Chance: Refbit 1: 0 setzen, → zweite Chance } keine Unterscheidung
Sache der MMU 0: Ersetzungskandidat } r/w - Zugriff
LRU enhanced 2nd Chance: modify / dirty-bit mitnutzen:
(R/D): (0/0) beste Wahl
(0/1) keine schlechte Wahl (nur geschrieben)
(1/0) keine gute Wahl (nur gelesen)
(1/1) schlechteste Wahl
Freiseitenpuffer: nutzt Vorabruf durch Schwellwerte gesteuert
low: ggf. Seiten auslagern
high: ggf. Seiten wieder einlagern (Vorausladen
Freiseiten kommen in Zwischenspeicher, bevor swap out
Seitenüberlagerung: lokal: nur Seiten des Prozessadressraums betroffen (Trap) } Seitenfehler
global: alle Seiten betroffen (Interrupt)
Thrashing verhindern: Arbeitsmenge bestimmen (ähnlich LRU) Seitenalter (Refbits) und diese komplett vorladen / aktiv lassen
Prozesse mit unvollständigen Arbeitsmengen stoppen / suspendieren → Long term Scheduling

DATEISYSTEME

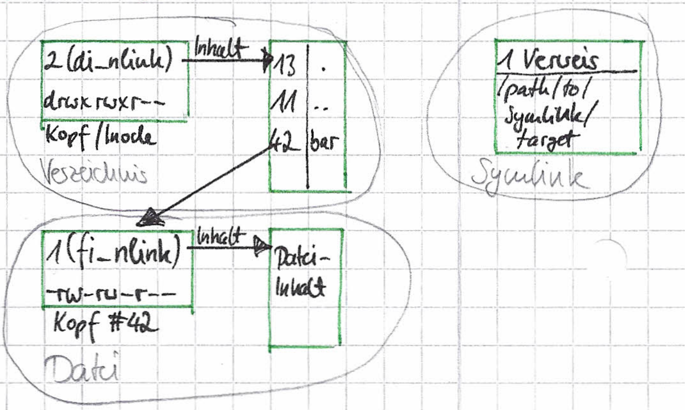
Berkeley Fast File System (BSD 4.2) Blockorganisation; Ext 2 analog, Zylinder → Blöcke

Journal-FS: Log (Re-Undo)
Copy-on-write (Log-structured FS; schreiben immer ans Ende / gute Schreibeffizienz; Änderungen Teil von Transaktionen
Auslesen vom Log bei booten → Inkonsistenzen vermeiden

NTFS:
Masterfiletable: Länge, Standard-Attribute, Zeitstempel, Zahl der Hardlinks
Sequenznummer, Dateiname (mehrfach bei vielen hardlinks)
Zugriffsrechte, Daten;
ggf. weitere Streams: Extends, Attributliste, Streams mit beliebigen Daten

	Linux	NTFS
Attribute	Inode	Stream in MFT
Data	Data blocks	kurze direkt in MFT lange in Clustern

VERZEICHNISSE



INODE

size (Bytes), Device ID, uID, gID, Typ (reg, FIFO), Rechte (r, w, x; Eigentümer, Gruppe, Welt)
Zeitstempel (inode change, last modified, last access)
hardlinkcount,
Zeiger auf Datenblöcke mit Dateiinhalt
Inode#: Index in der Tabelle von Dateiköpfen (nicht im Inode)

DIE

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

DATEI AUSLESEN

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

VERZEICHN

```c
DIR *dir = opendir("path"); struct dirent *entry;
if(!dir) 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: %d bytes\n", path, info.st_size);
}
if(errno != 0) die("readdir");
closedir(dir); // ggf. != 0 => die
```

LSTAT

lstat schaut Links an
stat geht hinter her

THREADS

```c
pthread_t thread;
int result = pthread_create(&thread, NULL, thread_start, argument); // (void *) / NULL
if(result) die("pthread_create");
result = pthread_detach(thread);
if(result) die("pthread_detach");
static void *thread_start(void *unused);
```

FORK + CHILD COLLECT

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

SERVER SOCKET

```c
int listenSock = socket(AF_INET6, SOCK_STREAM, 0);
if(listenSock == -1) die("socket");
struct sockaddr_in6 address;
adress.sin6_family = AF_INET6;
adress.sin6_port = hton(port); // 4711>0 atoi()
adress.sin6_addr = in6addr_any;
if(bind(listenSock, (const struct sockaddr *) &address,
        sizeof(address)) == -1){
    die("bind");
}
if(listen(listenSock, SOMAXCON) != 0){
    die("listen");
while(1){
    int clientSock = accept(listenSock, NULL, NULL);
    if(clientSock == -1){
        die("accept"); perror("accept");
        continue;
    }
close(listenSock); // ggf. != 0 => die(_);
```

CLIENT CONNECT

```c
struct addrinfo gai_hints;
struct addrinfo *gai_result, *server;
int gai_return, server_socket
FILE *server_fd;
memset(&gai_hints, 0, sizeof(gai_hints));
gai_hints.ai_family = AF_UNSPEC;
         .ai_socktype = SOCK_STREAM;
         .ai_flags = AI_ADDRCONFIG|AI_NUMERICSERV
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; // successfull connected
    if(close(server_socket) != 0){
        perror(..);
    }
}
freeaddrinfo(gai_result);
if(server == NULL){
    die("no server");
}
```

MAKEFILE

```make
CC=cc, CFLAGS=-std=c99 -pedantic, RM= rm -f
.PHONY: all clean
all: msgpipe
clean:
	$(RM) msgpipe *.o
msgpipe: msgpipe.c
	$(CC) -o $@ $(CFLAGS) -o $@ $^
```

READ FROM FILE *

```c
server_fd = fdopen(server_socket, "r"); // "w"
if(server_fd == NULL){
    close(server_socket);
    die("fdopen");
}
int result = fscanf(server_fd, "%d", &value);   (durchgestrichen)
char readData[5];
if(!fgets(readData, 5, server_fd)){
    if(ferror(server_fd)){
        die("_");
    }
fclose(server_fd);
```

WRITE

```c
fprintf(fd, "HELO %s\r\n", FQDN);
```

HANDLE SIGCHLD

```c
struct sigaction action;
sigemptyset(&action.sa_mask);
action.sa_handler = handleSIGCHLD;
action.sa_flags = SA_RESTART|SA_NOCLDSTOP;
sigaction(SIGCHLD, &action, NULL);
```

WAIT PASSIV SLEEP

```c
sigset_t mask, oldmask;
sigemptyset(&mask);
sigaddset(&mask, SIGCHLD);
sigprocmask(SIG_BLOCK, &mask, &oldmask);
while(childCount >= 16){
    sigsuspend(&oldmask);
}
sigprocmask(SIG_SETMASK, &oldmask, NULL);
```

HANDLER FOR ZOMBIES

```c
int errnoBackup = errno; int event;
pid_t pid;
while((pid = waitpid(-1, &event, WNOHANG)) > 0){
}
} errno = errnoBackup;
```

DUP

```c
int sock = ..;
int sock2 = dup(sock);
if(sock2 == -1) die("..");
fdopen(sock, "r"); fdopen(sock2, "w"); // w+ // a logfile
int fd1 = open("path", O_RDONLY);
if( == -1) => die(_);
if(dup2(fd1, STDIN_FILENO) == -1){
    die("dup2");
}
int fd2 = open("path", O_WRONLY|O_CREAT|O_TRUNC|
               S_IRUSR|S_IWUSR|S_IRGRP);
if(dup2(fd2, STDOUT_FILENO) == -1){
    die("dup2");
}
```

DUP2

EXEC

```c
execlp("perl", "perl", path, NULL);
execl(filename, filename, NULL);
```

P+V SEMAPHORE

```c
pthread_mutex_t mutex; int v = 0;
pthread_cond_t cond;
pthread_mutex_init(&mutex, NULL); // ret val: errno
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);
```

RING-PUFFER

```c
typedef struct BNDBUF{
    size_t write_pos, read_pos, entry_count;
    SEM *free_space, *filled_space
    int *entries; } BNDBUF;
BNDBUF *bbCREATE(size_t size){
    BNDBUF *newBuffer = malloc(sizeof(*newBuffer));
    newBuffer->entries = malloc(size * sizeof(*->entries));
             ->free_space = semCreate(size);
             ->filled_space = semCreate(0);
             ->write_pos = 0;
             ->read_pos = 0;
             ->entry_count = size;
    return newBuffer;
}
void bbPut(BNDBUF *bb, int value){
    P(bb->free_space);
    bb->entries[bb->write_pos] = value;
    ->write_pos = (bb->write_pos+1) % ->entry_count;
    V(bb->filled_space);
}
int bbGet(BNDBUF *bb){
    P(->filled_space);
    int value;
    size_t pos, new_pos;
    do{
        pos = bb->read_pos;
        value = ->entries[pos % ->entry_count];
        new_pos = pos+1;
    }while(!__sync_bool_compare_and_swap
           (&bb->read_pos, pos, new_pos));
    V(->free_space);
    return value;
}
```

Error-handling! semDestroy free

SYNCHRONISATION

einseitig → unterdrückend (Unterbrechung, Fortsetzung, Verdrängung)
mehrseitig → blockierend / nicht blockierend
(Schlossvar., Bedingungsvar., Semaphor) *CAS

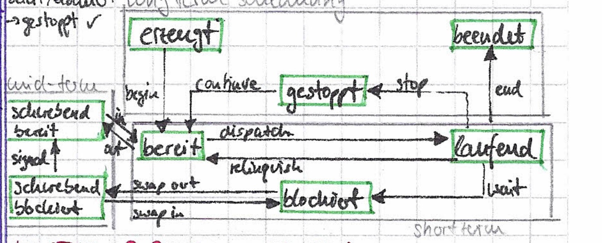
BM → wiederverwendbar → teilbar / unteilbar
BM → konsumierbar

PROGRAMM

Programm: Folge von Anweisungen
Prozess: Programm in Ausführung (abstrakt)
Prozessinkarnation: konkrete Ausführungsumgebung für ein Programm

PROZESSZUSTÄNDE

erzeugt: besitzt noch nicht alle nötigen BM
bereit: bereit zum Laufen (hat alle BM außer CPU)
laufend: wird gerade ausgeführt
blockiert: wartet auf Ereignisse (I/O, BM, Nachricht)
beendet: terminiert, noch nicht alle BM freigegeben (Exitstatus)
bereit/blockiert → gestoppt ✓



UNTERBRECHUNGEN

Trap: synchron, vorhersagbar, reproduzierbar, programmierbar (SEGFAULT, PAGEFAULT, illegale Anweisung)
Interrupt: asynchron, unvorhersagbar, nicht reproduzierbar (Tastendruck, DMA, E/A beendet, Seitenfehler bei globaler Ersetzungsstrategie)
Wiederaufnahmemodell: Fortsetzung; Interrupt muss, Trap kann
Beendigungsmodell: Trap kann, Interrupt darf nie

ARBEITSSPEICHER I

physik. Adressraum: (Hardware) hat Löcher, eindeutig
logischer Adrs.raum: (Kompiler, Binder, BS) alle Adressen gültig, mehrdeutig, abstrahiert vom Aufbau/Struktur des Hauptspeichers
virtueller Adrs.raum: (BS) abgebildeter log. Adr.raum, mehrdeutig, erlaubt Ausführung unvollständig im RAM liegender Programme
Programm → logischer (→ virtueller) → physikalischer

ADRESSRAUMSCHUTZ

Abteilung: Schutzgatter trennen BS und Programm, ggf. Schutzgatterregister statisch zur Programmausführung, physikalischer Adr.raumschutz, ein Prog. Betrieb
Eingrenzung: Begrenzungsregister pro Programm, mehr Prog. Betrieb, MPU CPU prüft Adressen, Überlappen der Bereiche möglich
Segmentierung: Basis/Längenregister bilden logischen Adr. Raum, Schutz dynamisch zur Laufzeit, Lader bestimmt physik. Adr.

ECHTZEITSYSTEME

weich: verspätete Ereignisse nutzbar, Terminverletzung tolerierbar
fest: " — " werden verworfen, " — " Ergebnis verworfen
hart: Katastrophe, " — " NICHT tolerierbar

EINPLANUNGSVERFAHREN

cooperative: Systemaufrufe nötig um Prozesse zu wechseln, Monopolisierung der CPU möglich
preemptive: P wird CPU entzogen → CPU-Schutz
deterministisch: CPU-Burstlängen und Termine bekannt, genaue Vorhersage der CPU-Auslastung möglich
probabilistic: nur Abschätzung möglich
offline: vor der Programmausführung → strikte Echtzeitsysteme
online: während " — "
FCFS: kooperativ, FIFO, Konvoi (kurze CPU-bursts nach langen)
RR: Zeitscheibe pro P (periodische Unterbrechungen) CPU-Schutz
Problem: kurze Stöße folgen langen, E/A-intensive P benachteiligt
VRR: P werden nach E/A bevorzugt eingeplant → bessere Verteilung der CPU-Zeit
SPN: kooperativ, a priori Wissen über Laufzeit, Verhungern möglich
SRTF: ereignisbedingte Umplanung (E/A), Verdrängung
HRRN: Einplanung nach erwarteter Bedien- und aktueller Wartezeit, beugt Verhungern vor
MLQ: P nach Typ eingeplant mit lokalen Strategien, stat/dyn globale Einplanung zwischen den Listen
FB: kurze/interaktive P bevorzugt, prio based Verfahren
Feedback, Erweiterung von RR
Kooperativ: FCFS, SPN — probabilistic: SRTF, HRRN
preemptiv: RR, VRR — mehrstufig: MLQ, FB (MLFQ)

KRITISCHE ABSCHNITTE

- Interrupts abschalten: cli / sti
- NPCS (non-preemptive-critical-section) enter/leave, Verdrängungssperre
- Semaphore P/V, Zähl-/Sperr-/u. Bedingungsvariable
- Bedingungsvariable: im KA warten ohne diesen zu blockieren
  - ↳ await: lässt P auf BV warten und gibt KA frei, bei Signal erneutes Betreten des KA
  - ↳ cause: löst mit der BV verknüpftes Ereignis aus, deblockiert wartende P

SEMAPHOREN

binäre SEM: verwaltet genau ein BM (gegenseitiger Ausschluss) ∈[0,1] vergibt unteilbare BM an P
zählende SEM: verwaltet mehr als ein BM (BM Instanzen selben Typs) ∈[0,N] vergibt konsumierbare + teilbare BM an P
Abstrakter Datentyp zur Signalisierung von Ereignissen zwischen zwei gleichzeitigen P, nicht-neg. Ganzzahl/P+V atomar

NEBENLÄUFIGKEIT

Interrupts, preemptive scheduling, Threads auf Multi CPU System

STR-FUNKTIONEN

```c
strtok(line, " \n\r") = char *first;
char *second = strtok(NULL, " \n\r");
char *saveptr;
strtok_r(line, " ", &saveptr);
strtok_r(NULL, " ", &saveptr);
strcat(str, str);
strcpy(*dest, *source);
```