Server: Socket erstellen

```
int listenSock=socket(AF_INET6);
if(listenSock<0) die("socket");
struct sockaddr_in6 addr={
    .di_family=AF_INET6,
    .di_addr=in6addr_any,
    .di_port=htons(port),
};
int flag=1;
if( setsockopt(listenSock,
  SOL_SOCKET,SO_REUSEADDR,&flag,
  sizeof(flag))<0) die("setsockopt");
if(bind(listenSock,&addr,sizeof(addr)
  <0) die("bind");
if(listen(listenSock,SOMAXCONN)<0)
    die("listen");
```

Verbindung annehmen:

```
while(1){
    int clientSock=accept(listenSock,NULL,NULL);
    if(clientSock<0){
        perror("accept");
        continue;
    }
    //do sth.. bsp.: bb_put(bb,clientSock);
    close(clientSock);
}
```

Client: Socket erstellen, DNS-Anfrage

```
char *host,*port;
struct addrinfo hints={
    .ai_socktype=SOCK_STREAM,
    .ai_family=AF_UNSPEC,
    .ai_flags=AI_ADDRCONFIG,
};
struct addrinfo *head;
int err=getaddrinfo(host,port,
    &hints,&head);
if(err!=0){
    if(err==EAI_SYSTEM){
        perror("getaddrinfo");
    }else{
        fprintf(stderr,"getaddrinfo
          %s\n",gai_strerror(err));
    }
}
struct addrinfo *curr;
int sock;
for(curr=head;curr!=NULL;curr=curr->ai_next){
    sock=socket(curr->ai_family,
      curr->ai_socktype,curr->ai_protocol);
    if(sock<0)continue;
    if(connect(sock,curr->ai_addr,
      curr->ai_addrlen)==0)break;
    close(sock);
}
freeaddrinfo(head);
if(curr==NULL) exit(EXIT_FAILURE);
```

Server: Lesen und Schreiben

```
while(1){ //deskr. holen
    int clientSock=bbGet(bb);
    int clientSockCpy=dup(clientSock);
    if(clientSockCpy<0){
        perror("dup");
        close(clientSock);
        continue;
    }
    FILE* rx=fdopen(clientSock,"r");
    //Files zum Lesen/Schreiben
    if(rx==NULL){
        close(clientSockCpy);
        continue;
    }
    FILE* tx=fdopen(clientSockCpy,"w");
    if(tx==NULL){
        close(clientSock);
        fclose(rx);
        continue;
    }
    handle_request(rx,tx); // do the work:
    if(close(rx)<0) perror("close");
    if(close(tx)<0) perror("close");
}
```

Client öffnen

Zeilen lesen von Client

Server: (Anzahl verarbeitet)

```
char buffer[MAX_LINE_LEN+1];
char *safe;
if((fgets(buffer,sizeof(buffer),rx)
  ==NULL)) return -1;
char *var;
char *nach;
if((Var=strtok_r(buffer,
  "delimiter",&safe))==NULL)
    return -1;
if(nach=strtok_r(NULL,
  "\n",&safe)==NULL)
    return -1;
if(strcmp(var,"sth"){
    //do sth
}
return -1;
```

Verzeichnis nach file/dir durchsuchen:

```
DIR *dir=opendir(path);
if(!DIR) die("opendir");
struct dirent *dir_entry;
struct stat buf;
while(errno=0,(dir_entry=readdir(dir))
  !=NULL){ //Fehlerfall: ?
  if((lstat(dir_entry->d_name,&buf)
    <0)||!S_ISREG(buf.st_mode)){
      continue;
  }
  //do sth.(ausgeben,chars zählen...)
}
if(closedir(dir)<0) perror("...");
```

Wert aus Buffer lesen an stelle path:

```
Static const char* cache_get(const
  char* path){
  Struct cache* curr=head;
  while(curr){
    if(strcmp(curr->path,
      path)==0)
        return curr->data;
    curr=curr->next;
  }
  return NULL;
```

Schreiben in Buffer:

```
Static void cache_put(const
  char* path, const char* d
{
  struct cache* x=
    malloc(sizeof(*x));
  if(!x)die("...");
  x->path=strdup(path)
  if(!x)die("...");
  x->data=data;
  do{
    x->next=head;
  }while(!CAS(&head,
    &x->next,x));
}
```

Globale Variablen definieren:

```
static struct bbuf *bb;
static volatile int server_dar;
static volatile int listen_sock;
static struct cache *head;
static volatile int clientCounter=0;
```

Richtige Anzahl an Argumenten übergeben:

```
if(argc!=2) die("usage");
```

Gelesenen String parsen:

```
long x=strtol(string,&endptr,10);
if(endptr!=NULL)die("invalid
    number");
```

execvp:


```
int execvp(const char
  *file,char *const
  argv[]);
```

qsort: mit struct

```
static compar(const void
  *p1,const void *p2){
  name *n1=(name*)p1;
  name *n2=(name*)p2;
  int a=strcmp(n1->
    nachname,n2->
    nachname);
  if(!a) return strcmp
    (n1->vorname,n2->
    vorname)
  return a;
}
Void qsort(void *array,
  size_t numbElements,
  size_t sizeElements,
  compar);
```

qsort: ohne struct

```
Static int compar(const
  void *a,const void *b){
  const char **p1=
    (const char**)a;
  const char **p2=
    (const char**)b;
  return strcmp(*p1,*p2);
}
```

strncmp:

```
Strncmp(const char *s1,
  const char *s2,size_t n)
-> 1 u. s2 ist zur
  n-ten Stelle verglichen
```

fnmatch:

```
int fnmatch(const
  char *pattern,const
  char *string,int flags)
=0 wenn matcht
```

Prozesse starten:

```
for(int i=0;i<ANZ_PROZ;i++){
    pid_t pid=fork();
    if(pid==-1){
        die("fork");
    }else if(pid==0){
        //Kindprozess: macht die
        Arbeiten, z.B. execvp(programm,
        oder fdopen(clientSock,"r");
    }else{
        //Eltern: gegebenenfalls close,
        sig.behandlung, zombies
        in pid... sammeln
    }
}
```

Threads starten:

```
pthread_t tids[ANZ_THREADS];
for(int i=0;i<ANZ_THREADS;i++){
    errno=pthread_create(&tids[i],NULL,
      &kleinebereich,NULL);//also thread-
      create, pid, Argumente
    if(errno!=0)die("pthread_create");
}
errno=pthread_join(tids[i],NULL);
if(errno...){//threads warten ungewünschter
errno=pthread_detach(tids[i]);
if(errno...){// threads warten NICHT
    aufeinander
}
=> nur 1 von beiden!
```

Ring-puffer/bounded buffer:

```
typedef struct BNDBUF {
    SEM *semput,*semget;
    int *buffer;
    size_t _Atomic r;
    size_t w;
    size_t length;
}BNDBUF;
```

```
BNDBUF *bbCreate(size_t size){
    BNDBUF *ring;
    ring=malloc(sizeof(BNDBUF));
    if(ring==NULL){
        free(ring);
        return -1;
    }
    ring->semget=SemCreate(0);
    if(ring->semget==NULL){
        return -1;}
    if(SIZE>(size>INT_MAX){
        free(...);return -1;}
    ring->semput=SemCreate
        (size);
    if(ring->semput==NULL){
        free(ring->semput);
        free(ring->semget);
        free(ring);
        return -1;
    }
    ring->length=size;
    ring->r=ATOMIC_VAR_INIT(0);
    ring->w=0;
    ring->buffer=malloc(size);
    if(ring->buffer==NULL){
        free((void*)ring->buffer);
        free(//alle andere auch);
        return -1;
    }
    return ring;
}
```

```
void bbDestroy(BNDBUF *bb){
    if(bb!=NULL){
        SemDestroy(bb->semput);
        SemDestroy(bb->semget);
        free(bb);
    }
}
void bbPut(BNDBUF *bb,int value){
    P(bb->semput);
    int expected;
    int desired;
    do{
        expected=atomic_load(&bb->w);
        desired=(expected+1)%bb->length;
        bb->buffer[expected]=value;
    }while(atomic_compare_exchange_
        strong(&bb->w,(int_Atomic*)
        &expected,desired));
    V(bb->semget);
}
int bbGet(BNDBUF *bb){
    P(bb->semget);
    int expected;
    int desired;
    do{
        expected=atomic_load(&bb->r);
        desired=(expected+1)%bb->length;
        output=bb->buffer[expected];
    }while(atomic_compare_exchange_
        strong(&bb->r,(int_Atomic*)
        &expected,desired));
    V(bb->semput);
    return output;
}
```

Bounded Buffer initial:

```
bb=bbCreate(size);
if(!bb)die("bb");
```

Semaphor:

```
typedef struct SEM {
    pthread_mutex_t m;
    pthread_cond_t c;
    int val;
}SEM;
SEM* semCreate(int val){
    SEM *sem;
    sem=malloc(sizeof(SEM));
    if(sem==NULL) return NULL;
    sem->val=initval;
    errno=pthread_mutex_
        init(&sem->m,NULL);
    if(errno!=0){
        free(sem);
        return NULL;
    }
    errno=pthread_cond_
        init(&sem->c,NULL)
    if(errno!=0){
        free(sem->m);
        free(sem);
        return NULL;
    }
    return sem;
}
void semDestroy(SEM *sem){
    if(sem!=NULL){
        free(sem->c);
        errno=pthread_mutex_
            destroy(&sem->m);
    }
}
Void V(SEM *sem){
    //increment
    pthread_mutex_lock(
        &sem->m);
    sem->val++;
    pthread_cond_signal
        (&sem->c);
    pthread_mutex_
        unlock
        sem->m);
}
Void P(SEM* sem){
    pthread_mutex_lock
        (&sem->m);
    if(sem->val<=0){
        while(sem->val<=0){
            pthread_cond_wait
            (&sem->c,&sem->m);
        }
        sem->val--;
    }else{
        sem->val--;
    }
    pthread_mutex_unlock
        (&sem->m);
}
```

Signal handler:

```
void handle_sigint(int
  signal){
    int old=errno;
    close(listen_sock);
    errno=old;
}
```

Signalbehandlung initialisieren:

```
Sigset_t int_Set;
Sigemptyset(&int_set);
Struct Sigaction sig_int={
    .sa_handler=handle_sigint,
    .sa_mask=int_Set,
    .sa_flags=SA_RESTART,
};
if(sigaction(SIGINT,&sig_int,NULL)<0)
    die("sigaction");
Struct sigaction sig_pipe={
    .sa_handler=SIG_IGN,
};
Sigemptyset(&sig_pipe.sa_mask);
if(sigaction(SIGPIPE,&sig_pipe,NULL)<0)
    die("sigaction");
```

Signal blockieren (kurzzeitig):

```
sigset_t old,new;
sigemptyset(&new);
sigaddset(&new,SIGCHLD);
sigprocmask(SIG_BLOCK,&new,&old);
//do sth. während sig. blockiert ist
sigprocmask(SIG_SETMASK,&old,
  NULL);
```

bei do sth.: auf ing. ein Sig. warten, hier vorher grundsätzl. while(globpid==-1) Sigsuspend(&old);

Handler für Sigsuspend:

```
Static void handler(int sig){
    int err=errno;
    int event;
    while(1){
        pid_t pid=waitpid(
            -1,&event,
            WNOHANG);
        if(pid<=0) break;
        if(pid==globpid&&
            WIFEXITED(event)){
            globpid=pid;
            globevent=event;
        }
        //hier prüfen ob Vorgängerprozess
        gestorben ist, wenn möglich:
        errno=err;
    }
```

Main-Funktion:

```
int main(int argc,
  char *argv[]){
    //do sth.
}
```

Sigprocmask: bei mehreren Threads: nicht sigprocmask sondern pthread_sigmask();

```
Prozs.: kill(pid,sig);
Thread: pthread_kill(thread,
    sig);
```

In stdout schreiben:

```
sprintf(stdout,"status:
  success %d",success);
```

→ Ende Main: fflush(0);

FFLUSH(); am Ende der Main! (wenn Stream)

Bei mehrfädigen Weiterleitungen muss SIGPIPE durch die Signalbehandlung ignoriert werden!

Signale:

SIGCHLD: Child Stopped/terminated
SIGINT: Interrupt (keyboard)
SIGKILL: kill signal
SIGPIPE: Broken pipe: write to pipe without readers
SIGSTOP: Stop proz.
SIGSEGV: invalid memory reference
SIGTERM: terminate signal

Ringbuffer Variante:

```
int bbPut(BNDBUF *bb,int value){
    int expected;
    int desired;
    do{
        expected=atomic_load(&bb->w);
        desired=(expected+1)%bb->length;
        if(desired==bb->r) return -1;
    }while(atomic_compare_exchange_
        strong(&bb->w,(int_Atomic*)
        &expected,desired));
    bb->buffer[expected]=value;
    V(bb->semget);
    return 0;
}
```

Ringbuffer Variante:

```
int bbGet(BNDBUF *bb){
    int expected;
    int desired;
    do{
        expected=atomic_load(&bb->r);
        desired=(expected+1)%bb->length;
        if(expected==bb->w) return -1;
        output=bb->buffer[expected];
    }while(atomic_compare_exchange_
        strong(&bb->r,(int_Atomic*)
        &expected,desired));
    V(bb->semput);
    return output;
}
```

Semaphor Initialisierung:

```
full=semput=
    =semCreate(SIZE);
free=semget=
    =semCreate(0);
//Fehlerbehandlung!
```

CAS-Schleife:

Atomic-Funktionen:

Lesen: int atomic_load(volatile A *obj);
=) int y=atomic_load(&x)
= int y=x;

Schreiben: void atomic_store(volatile A *obj, C value);
=> atomic(&x,42);
= x=42;

Initialisierung:
ATOMIC_VAR_INIT(value)
=> _Atomic int x=ATOMIC_VAR_INIT(42);
= int x=42;

Warten bis Anzahl Prozessen erreicht:

```
if(activeProc>=PROCS_MAX)
    Sigsuspend(&...);
activeProc++;
sigset_t empty;
Sigemptyset(&empty);
if(sigprocmask(SIG_BLOCK,
  &old,NULL)<0)die(...);
if(sigprocmask(SIG_UNBLOCK,
  &old,NULL)<0)die(...);
```

Close-on-exec Flag:

```
Offene Filedeskr.: fork → vererbt | exec → bleibt
→ unterbinden:
if(fcntl(fd,F_SETFD,FD_CLOEXEC)<0)
    die("fcntl");
Wenn dup/2:
int flags=fcntl(fd,F_GETFD,0);
if(flags<0) die("fcntl")
//dup/dup2
if(fcntl(fd,F_SETFD,flags,FD_CLOEXEC)<0)
    die("fcntl");
```

bbGet besser gegen ABA-Problem geschützt:

```
void *bbGet(BNDBUF*bb){
    void *value;
    size_t pos;
    size_t nextPos;
    P(bb->semget);
    do{
        pos=atomic_load(&bb->r);
        size_t index=pos%bb->size;
        value=bb->data[index];
        if(pos<SIZE_MAX){
            nextPos=pos+1;
        }else{
            nextPos=index+1;
        }
    }while(!CAS(&bb->r,pos,
        nextPos));
    V(bb->semput);
    return value;
}
```

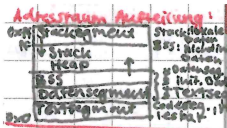
SIGPIPE Behandlung -andere Variante:

```
sigset_t ma;
if(-1==sigemptyset(&ma))die(...);
if(sigaddset(&ma,SIGPIPE)==-1)die(...);
struct sigaction act={
    .sa_handler=SIG_IGN,
    .sa_mask=ma,
    .sa_flags=SA_RESTART,
};
if(0>sigaction(SIGPIPE,&act,NULL)==-1)
    die(...);
```

Makefile:

```
.PHONY:all clean
CC=gcc
CFLAGS=-std=c11 -pedantic
    -Wall -Werror -D_XOPEN
    _SOURCE=700
all: sister
clean:rm -f request-http.o
    connection-fork.o sister.o
    sister
%.o:%.c connection.h request.h
    cmdline.h ...h httools.h
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $<
sister: sister.o con...o con.h req.o req.h
    ...h httools.o ...h...h ...
    $(CC) $(CPPFLAGS) $(CFLAGS) -o $@ $^ sister
```

Adressraum Aufteilung:



Page fault Ablauf:

1. MMU löst Trap aus
2. BS (SW Handler für Page Fault) aus
3. Freier Platz im Hauptspeicher wird gesucht; ggf. andere Seite auslagern
4. Einlagerung wird von Platte angefordert
5. Prozesszustand blockiert
6. Interrupt kommt u. bemerkt, dass Einlagerung abgeschlossen ist
7. Present-Bit wird auf 1 gesetzt
8. Prozzustand bereit
9. Nach Einplanung (Prozzustand laufend) setzt der TLB angefordert Seite wird geladen

Page fault:

Prog. greift auf Speicherbereich zu, der sich nicht im Hauptspeich. befindet (ausgelagert...)

Minor page fault: Seite im HS, aber MMU hat Present-Bit noch nicht gesetzt → Eintrag für Seite in MMU machen

major page fault: Seite nicht im HS → Einlagern u. Present-Bit durch MMU setzen

invalid page fault: Zugriff auf Adr., die nicht Teil des virtuellen Adr.raums ist → Segmentation Fault

Trap (Synchron) / Interne Interrupt (Asynchron):

Ursache, die vorhersagbar, reproduzierbar, Behandlungsmodell z.B.: ungültige Adr., Unbekannter Befehl, Teilen durch 0, Systemaufrufe, Seitenfehler bei lokaler Ersetzungsstrategie

Ursache, unvorhersagbar, nicht reproduzierbar, Wiederaufnahme-Modell (Nebeneffektfrei!), z.B.: Beendigung I/O-Operation, Seitenfehler bei glob. Ersetz.strategie

Betriebsmittel:

- wiederverwendbar: durch mehrseitige Sync. anfordern, benutzen, freigeben
- verbrauchbar: durch einseitige Sync. anfordern, transient (=vergänglich), nach Freigabe zerstört z.B. Signal, Trap

Schichtenfolge in Rechensystemen:

5 Prog.sprach.ebene
4 Assembl.sprachebene
3 Maschin.prog.ebene
2 Befehlssatzebene
1 Mikroarchitekturebene
0 digitale Logikebene
2-3 Teilinterpret. durch BS

CC — **AS, Ld** — **UNIX** — **XPS-64** — **ARM**

Schlafzustand:

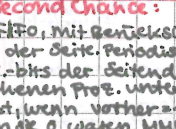
notwendig:
1. Ausschließlichkeit in der BM-Nutzung (Mutex)
2. Nachforderung eines ineinander BM (Hold&wait)
3. Unentziehbarkeit der zugeteilten BM (no preemption)

hinreichend: 4. zirkuläres Warten ist eingetreten

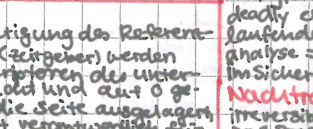
Second Chance:

FIFO, mit Berücksichtigung des Referenzbits der Seite. Periodisch (Zeitgeber) werden Ref.-bits der Seitendeskriptoren des unterbrochenen Prog. unterzogen und auf 0 gesetzt, wenn sie 0 waren. MMU ist verantwortlich für setzen der Ref.-bits → ausgewählt wird die global zuerst eingelagerte, unreferenzierte Seite

Paging einstufig:

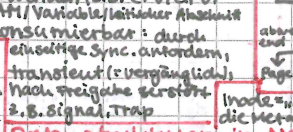


Paging zweistufig:



Inode = „Systeminterne Struktur die Metadaten einer Datei enthält"

Datenstrukturen im Namensraum (Datei-Systeme):



Verklemmung:

beteiligte Prozesse warten auf Eintritt von Bedingungen, die nur durch andere Prog. (die ebenfalls warten) hergestellt werden können) ggf. untätig. Chancen erkannt werden, falls Ereignis bekannt) ⇒ deadly embrace im

Verklemmungsvorbeugung:

1. nicht block. Sync. (Atomaritäten auf tieferer Ebene nutzen)
2. alle benötigten BM unterbar anfordern
3. BM virtualisieren, damit Entzug der realen BM ermöglichen

Verklemmungsvermeidung:

Voraussagen zu Prozess- und BM-Anforderungen nötig, Vermeidung von deadly embrace durch laufende Anforderungsanalyse ⇒ BM-Zuteilung nur im sicheren Zustand

Nachtrag Verklemmung:

irreversibl Blockierung von >= 2 Prozz. → Prozessorsystem macht keinen Fortschritt

Seitenflattern

Phänomen der globalen Seitenersetzung (Menge residenter Seiten pendelt zu klein, kritisches Potenz haben, wenn die durch Seitenein-/-auslagerung verursachte E/A die Zusammenarbeit mit Prozessen stört (niert ⇒ Prozz. verbringen mehr Zeit beim Rechnen ⇒ Leistungseinbruch

Prozz.zustände: bereit → laufend → blockiert (da Page fault)

Umlaufsperre:

Gleichzeitige Prozz. kreisen ohne Nutzen, Schleifendauer bedeutet Wartezeit. Nutzarbeit startet mit atomarer Aktion z.B. (CAS), wechselseitiger Ausschluss (Software gesteuert umgesetzt) Verklemmungsgefahr, verhungern, interferieren mit Planer, nicht robust

Monitorvergleich:

Semantik Bedingungsvariablen: blockierend (gibt Signal + nehmer Vorrang), nicht blockierend (gibt Signal aber Vorrang)

Hansen: blockierende Bedingungsvariablen; Signalisierung lässt Signalgeber den Monitor verlassen, nachdem er die Signalnehmer bereit gesetzt hat; Neuauswertung der Wartebedingung nötig

Hoare: blockierende Bedingungsvariablen; -!!-...genau einen Signalnehmer fortfahren → atomare Aktion, Neuauswertung d. Wartebedingung entfällt

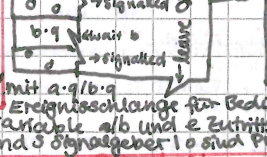
Mesa: nichtblockierende Bedingungsvariablen, -!!- lässt Signal GEBER im Monitor FORTFAHREN, nachdem er einen oder alle Signalnehmer bereit gesetzt hat

Monitor als ADT:

ADT dessen Zugriffsoperationen implizit synchronisiert sind, mehrseitige Sync. mittels Semaphor / Schloss-Variable c/ an Monitorschnittstelle, einseitige Sync. mittels Bedingungsvariable innerhalb des Monitors

Monitorwarteschlange: Prozessexemplare, die ein Monitor anwarten (warten, falls Monitor durch anderen Prozess belegt ist)

Ereigniswartesollange: Prozessexemplare, die im Monitor auf Aufhebung einer Wartebedingung warten



Journaling-File-System:

Protokoll der Änderungen an einer Datei wird in Protokolldatei (Logfile) geführt, jede Änderung wird als Transaktion verstanden (Erzeugen, Löschen, Umbenennen, ...). Vor jeder Aktion wird Logeintrag erstellt → bei Bootvorgang: Abgleich mit Festplatte; Redo / Undo: Inkonsistenzen vermeiden ⇒ Transaktionen werden entweder vollständig oder gar nicht ausgeführt, Nachteil: ineffizient

Speicherung von Daten:

Kontinuierliche ~: Datei in Blöcken mit aufsteigender Blocknummer gespeichert → minimale Positionierungszeit → Echtzeitsystem

Verkettete ~: Blöcke einer Datei sind verkettet → Datei kann wachsen, aber Speicher nur Verzeigerung nötig, häufiges Positionieren, Schlechter Direktzugriff

FAT-Ansatz: (File allocation table = Datei-Zuordnungstabelle) verkettete Speicherung, aber Verkettung in speziellen Blöcken gespeichert → Speicher nur Verzeigerung benötigt nicht Nutzspeicher, aber zusätzlicher Block muss geladen werden

Indiziertes Speichern: spezieller Platzenblock enthält Blocknummern der Datenblocks einer Datei → Nachteil: feste Anzahl von Blöcken in Inderblock

Interne Fragmentierung:

Verschnitt innerhalb des angeforderten Speicherblocks z.B. Paging → Lösung: Lochliste schlau sortieren

Externe Fragmentierung:

Verschnitt zwischen den angeforderten Speicherblöcken z.B. Segmentierung → Lösung: keine allg. Lösung außer teure Verschmelzung durch BS

Nichtblockierende Sync. mit CAS:

Prinzip: Atomares Prüfen ob Wert der Variable, die verändert werden soll mit dem erwarteten Wert übereinstimmt; wenn ja: Wert zu neuem ändern; Vorteil: keine Verklemmungsgefahr, gerade Parallelität; Nachteil: wenn CAS-Aktualität muss ganze Transaktion wdh. werden, komplizierte Implementierung, Verhungerungsgefahr; Anwendungsbsp.: bounded Buffer: Lesepointer vor Lesen mittels CAS überprüfen u. entsprechend ändern

Einseitige Synchronisation:

wirkt auf 1 beteiligten Prozz. logisch (geg. durch Reihenspiel der Prozz. oder bedingt (Wartebedingung, muss evtl. von anderem Prozz. ausgelöst werden) für Zugriff auf konsumierbare BM

Mehrseitige Sync.:

wirkt auf alle beteiligten Prozz. blockierend (wechselseitiger Ausschluss) oder nicht-blockierend

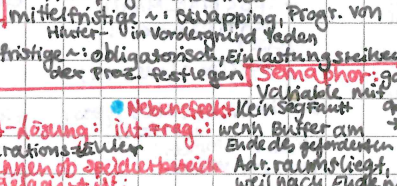
Sperrsatzbefehle = atomare Aktionen) für Zugriff auf unteilbares, gemeinsames BM mit wiederrotierenden Änderungen

Adressraumschutz:

durch Abteilung schutzgatterregister hält Adresse die Anwendungsprogr. von Monitor (BS) kennt. durch Eingrenzung: Begrenzungsregister legen ober-/untergrenze fest → eignet sich für mehrere Anwendungen auf 1 Hauptspeicher

Baddy:

Scheduling



Einplanungszeitpunkt:

begin: nach Prozz.erzeugung

relinquish: wenn Prozz. CPU freiwillig → kooperativ!

signal: falls das vom Prozz. erwartete Ereign. eingetreten

continue: sobald Prozz. wieder aufgenommen ist werden kann

Zustände:

erzeugt: fertig zur Programmerarbeitung

gestoppt: erwartet Fortsetzung/Beendigung

beendet: er wartet Entsorgung

schwebend bereit: Swap-out ist erfolgt, wartet auf Swap-in

schwebend blockiert: Swap-out ist erfolgt, wartet auf Ereigniseintritt, dann schwebend bereit

bereit: Prozz. auf Bereitliste, bereit zur Ausführung durch CPU

laufend: Zuteilung CPU ist erfolgt, vollzieht Rechenschritte

blockiert: wartet auf BM-Zuteilung ggf. durch anderen Prozess)

Arten:

langfristige ~: Lastkontrolle, Grad an Mehrprogrammbetrieb

mittelfristige ~: Swapping, Progr. von Hinter- in Vordergrund laden

Kurzfristige ~: obligatorisch, Einlastungsreihenfolge der Prozz. festlegen

Semaphor:

ganzzahlige Variable mit 2 unteilbaren Operationen V|P → einsatz: multilaterale Sync. von Kritt. Abschnitt durch mutual excl.

ABA-Lösung:

Generations-Zähler

Erkennen ob spezieller Speicherbereich ausgelagert ist:

MMU stellt dies bei der Umrechnung der virt. Adr. anhand des Present Bits fest.

Nebeneffekt:

iut. frag.: Kein Segment wenn Puffer am Ende des gefundenen Adr.raums liegt, weil nach Ende nach weiterer interner Verschnitt folgt

Last-wake-up-call:

Prozz. für den Wartebed. erfüllt, während belegt den krit. Abschnitt muss sich verhalten:
1. krit. Abschnitt freigeben, also verlassen
2. blockieren, bis anderer Prozz. Wartebed. auflebt
3. um Wiedereintritt bewerben

Problem: weiterhin krit. Abs.? hat 1), aber noch nicht 2) und P_2 hebt schon auf → nicht aus warteliste eint kümmern

Datei-Attribute unter NTFS:

NTFS: Das adr. u. -inhalt jeweils in eigenen Stream gespeichert, die sind alle im MFT-Eintrag vorhanden

Monitor-Exp. thread

Kapselung Zusammenhang -tion -zusammengezogene von Prozz. gemeins. Datei - Programmierer muss sich nicht um Sync.-art kümmern

Kusexe

-JoinsThread-Sync.; Bibliothek

-Cond_var