E 5 Echtzeitbetriebssystem/Realzeitbetriebssystem (RBS) am Beispiel OS9000 (OS9)

► Realzeitbetriebssystem (RBS): Bereitstellung von "Infrastruktur" zu Verwaltung des Prozessrechners (PZR).
► Viele Aufgaben sind dabei unabhängig vom technischen Prozess und für jeden neuen Einsatz des PZR nötig.
► Alle diese Aufgaben werden sinnvollerweise zusammengefasst, einmal realisiert und wiederverwendbar dem PZR-Programmierer zur Verfügung gestellt.
►Zusammenfassung im sog. **Realzeitbetriebssystem (RBS)**

►Aufgabe eines Realzeit-Betriebssystems (insbesondere des Schedulers,), den einzelnen Rechenprozessen zu jedem Zeitpunkt die prozessbedingte erforderliche Rechenzeit just-in-time Verfügung zu stellen.

E 5.1 Einführung

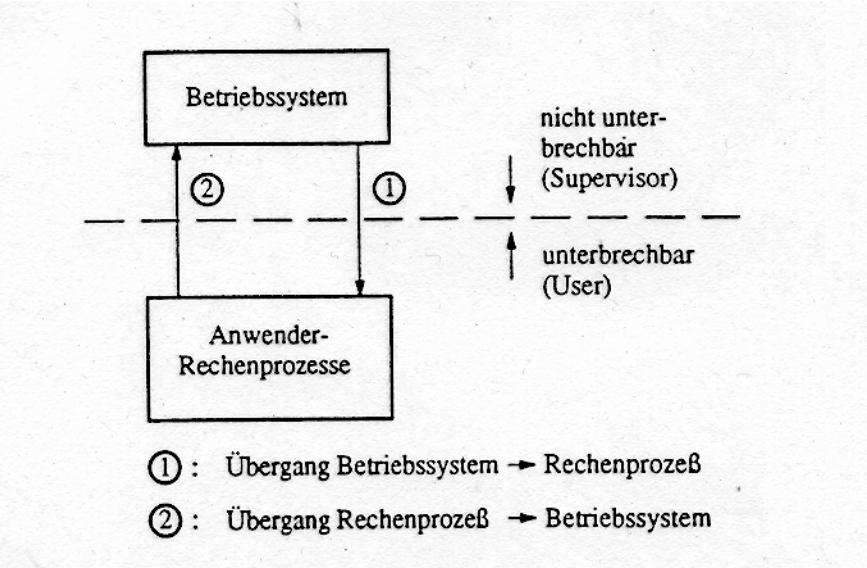
►RBS (wie andere Betriebssysteme (BS) auch) verwaltet und teilt gegebenenfalls den Tasks auf Anforderung die verfügbaren Betriebsmittel zu.

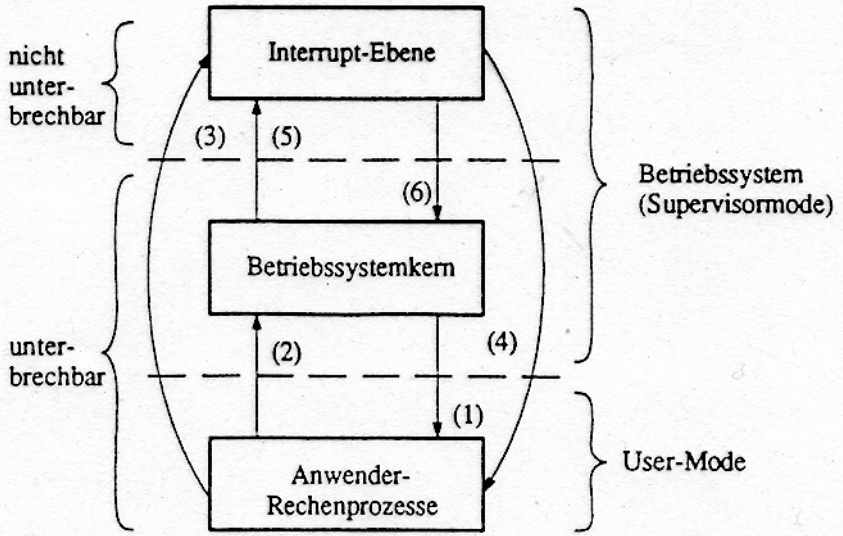
►Betriebsmittel Beispiele:

- CPU-Rechenzeit Zuteilung
- Arbeitsspeicher, Massenspeicher (Zugang, Platzverbrauch, Zugriffschutz)
- Ein/Ausgabegeräte
- Interprozess-Kommunikations- und Synchronisationsressourcen

Besondere zusätzliche Anforderung an das RBS ist die garantierte maximale Reaktionszeit auf Ereignisse

Forderung: RBS-Präemptiv (unterbrechbar) d.h. auch Ausführungen im RBS muss durch externe-Ereignisse unterbrechbar sein





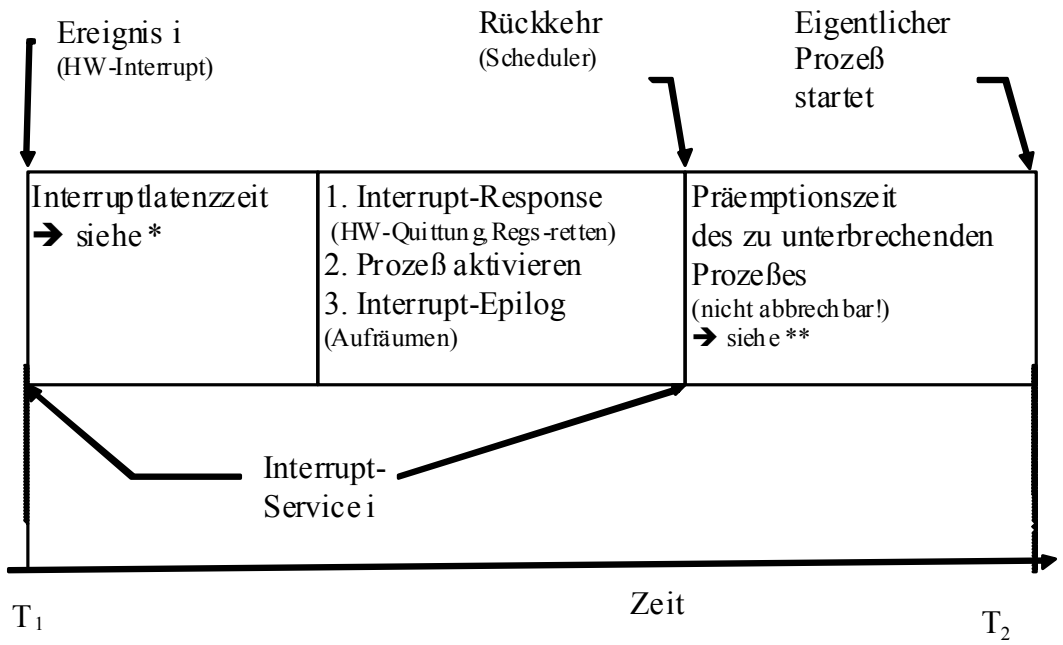
Struktur eines einfachen Betriebssystems /Färber/

Stuktur eines RBS /Färber/

Wie schnell ist die Raktion? Bestimmung bzw. Messung der Reaktionszeit des RBS in der Praxis:

Prozessereignis-Reaktionszeit des RBS

- **Ausgangslage:** Prozessereignis i tritt auf ➔ Interrupt i wird aktiviert ➔Task i (höchste Priorität) muss aktiviert werden, aber Task n läuft gerade (Interrupt sei erlaubt und unterbricht nun Task n).



* worst case: Maximum aus: längster Befehl des CPU-Befehlsvorrats und längste programmgesteuerte Interruptsperrzeit

** Wenn die zu unterbrechende Task eine RBS-Aufruf getätigt hat der die Umschaltung **blockiert und dadurch verzögert.** Z.B.: Scheduler

- **Prozessereignis-Reaktionszeit dauert worst-case von T_1 bis T_2**

  ➔ ´Käufersicht´ == Kontextswitchzeit im gezeigten Prozessverständnis

  ➔ ´Verkäufersicht´ == programmgesteuerter Wechsel von Task zu Task ohne Interrupt!
  Kontextswitchzeit == nur Präemptionszeit + Register laden.

  ➔ **´Interruptlatenzzeit´ bis zu Aktivierung der gewünschten Task´**

Weitere Eigenschaften typischer RBS:

►dynamische Prioritätensteuerung der Tasks

►RechenProzess (Task) -Synchronisation und -Kommunikation

►Aus-Praxis: RBS Flash-ROM fähig, skalierbar in der Größe, minimaler RAM Footprint, „in place execution“ von Tasks/Threads

►RBS stellt dem Anwender i.d.R. zwei Schnittstellen (SS) zur Verfügung:

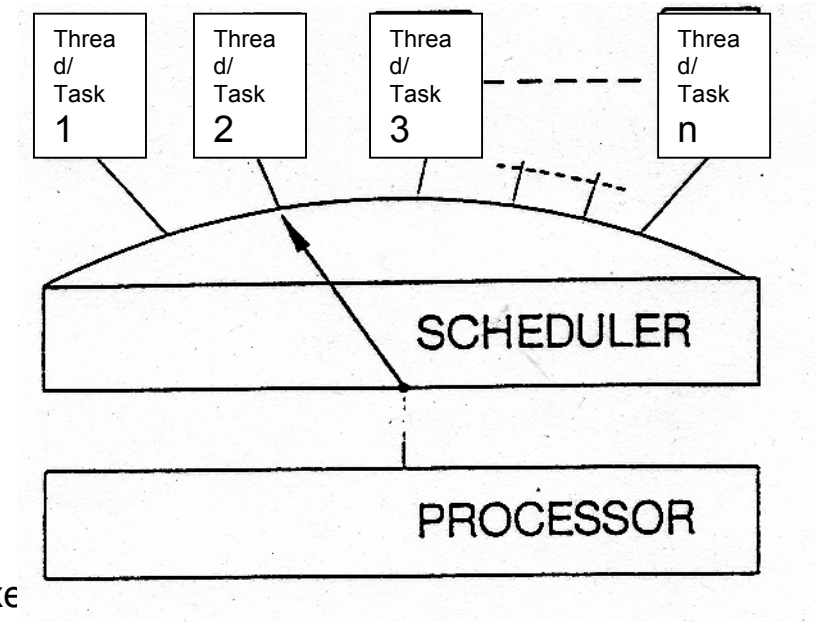
►Bedienoberfläche zur Bedienung und Steuerung des BS und der Tasks mittels RBS-Kommandos (command-shell) ► OS-9 mshell/shell

►RBS-Aufrufschnittstelle (*system-calls*) SYSA für den Anwendungsprogrammierer, der die Tasks/Threads entwickelt und von der Task aus die Vorgänge im und Dienstleistungen des RBS des PZR mittels Programmcode steuern und nutzen will. ►OS9000 z.B. via C-Library

E 5.2 Task- und Threadkonzept des RBS

E 5.2.1 Task- bzw. Threadmodell und Task-(Thread-)zustände

►Eine Task besteht nach Start aus einem Hauptthread und danach aus ggf. mehreren weiteren Threads und den Task- und Threadverwaltungsdaten.
►Eine Task ist also ein Container für Threads
►Ein Thread ist ein ‚Ablauffaden' (Programmcounter), der eigenständig Programmcode abarbeitet.
►Aller Thread-Programmcode befindet sich innerhalb der Task
►Alle Threads einer Task teilen sich den Datenbereich einer Task
►Jeder Thread besitzt einen eigenen Stack und Registersatz
► Der Begriff Hauptthread und Task wird oft gleich verwendet
►Tasks und Threads werden durch das RBS verwaltet.

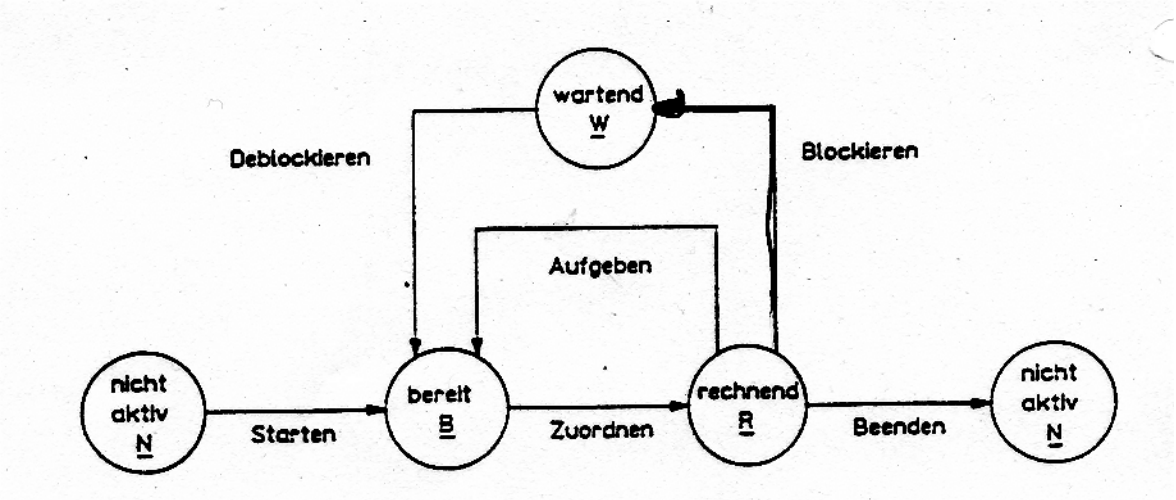


►Jeder laufender Thread ist eineindeutig anhand einer HauptthreadID (TaskID) bzw. ThreadID identifizierbar
►TaskID/ThreadID wird zum Startzeitpunkt vergeben und bei Beendigung gegebenenfalls neu verteilt ► Kommunikationsadresse zur Laufzeit.
► Os9000: *process_id unsigned short* oder *pthread_t*
► 16-bit => 65535 Tasks/Threads gleichzeitig möglich

►Threads laufen scheinbar parallel ab => in Realität : schnelle Umschaltung zwischen den einzelnen Tasks/Threads durch SCHEDULER = Threadmultiplexe

►Scheduler: Programmteil des RBS das immer durchlaufen wird wenn System-Calls durch eine Task getätigt werden und i.d.R. unmittelbar nach Abarbeitung von externen Ereignissen/HW-Interrupts in der ISR ►z.B. Timerinterrupt

Threadzustände (Taskstates) in denen sich eine Task/Thread befinden kann:



N: nicht aktiv (non-active), Programmcode nicht aktiv.

B: bereit (ready,active), eine oder mehrere Tasks/Threads warten auf Zuteilung der CPU durch SCHEDULER => **rechenwillig** >= 1 Task/Thread

R: rechnend (running), der eine Thread, der in einem Singleprozessorsystem!! die CPU besitzt!!

W: wartend (suspended,waiting) : eine oder mehrere Threads die auf das Eintreten eines Ereigniss warten. => nicht rechenwillig

E 5.2.2 Threadzustandwechsel

Vertagung (blockieren,suspend):

Zustandswechsel: R ►W (OS9000-Begriff sleeping)

Auslösung durch:
Fall A: Thread entscheidet durch Programmstatements selbstständig, dass sie die CPU nicht mehr benötigt bis ein bestimmtes Ereignis (event,signal,alarm,timeout) eintritt.
►OS9000:
_os_ev_wait() -event-wait;
_os_sleep (&time,...) -eine bestimmte Zeit time warten
time=0 bedeutet unendlich warten, wecken durch Empfang eines Signal , Alarms

Fall B: Thread fordert nächst möglichen Zugang zu exklusiv nutzbaren Betriebsmittel (**system resources)** vom **RBS** durch RBS-Dienstleistungsaufruf (**system-call) und RBS** stellt fest, dass das angeforderte Betriebsmittel anderweitig belegt ist .

►**RBS** blockiert Thread solange bis Betriebsmittel frei ist **und** reiht Thread in eine Warteschlange (**queue)** auf exklusive Zuteilung des Betriebsmittels ein.

►typischer Fall: konkurrierender quasi gleichzeitiger Zugriff mehrer Threads auf eine gemeinsame Festplatte (Zugriffszeit im 10msec Bereich) ►RBS muss die Zugriffe **serialisieren: der 1. darf** der 2. und folgende muss warten.

►OS9000: Device -I/O-/Thread-Condition-Variable – Queue

Fall C ➔Sonderfall OS9000 parent-task/-thread wartet auf die Beendigung einer seiner Kinder

Fortsetzung (deblocking):

Zustandwechsel: W ►B

Auslösung: durch:

war **Fall A**: Ereignis auf das ein Thread wartet ist eingetreten

►Sonderfall OS9000: !! Während der Thread auf das Auftreten eines Eventswertes wartet und ein Signal eintrifft, wird er vorzeitig geweckt W ►B obwohl nicht der Eventwert erreicht ist!!

war **Fall B**: Betriebsmittel wird frei, Thread bekommt Resource exklusiv zugeteilt und wird vom RBS als rechenbereit eingestuft

Verdrängung:

Zustandwechsel: R ►B

Auslösung durch:
►höherpriorer Thread wird auf Grund eines externen/internen Events durch das RBS in den Zustand bereit versetzt ►RBS entzieht dadurch dem unterbrochenen Thread den Prozessor.

►OS9000: Interrupt tritt auf ► Treiber/Thread löst Event aus auf das ein höher priorer Thread wartet oder der running-Thread setzt selbst einen Eventwert bzw. Signal ab, auf den ein höherpriorer Thread wartet

►zugeteilte Zeitscheiben (OS9000-Timesharing) des rechnenden Threads ist abgelaufen und:.

►OS9000: ► Praktikumssystem OS9000 alle 1 ms erfolgt ein Uhreninterupt, ein Tick der Zeitscheibe (TIC) wird alle 2ms weitergezählt

►Jeder Thread besitzt z.B. ein Kontingent an Vielfachen der Zeitscheiben (NICHT OS9000!!)

►spätestens nach Ablauf der TICs erhalten andere gleichpriore Threads eine Chance.

►OS9000:►Aging der wartenden Prozesse verändert deren relativen Prioritätsabstand zum running-Tread.

►Wechsel dann wenn höherpriorer Thread wartet.

Zuteilung:

Zustandswechsel: B ►R

Auslösung: RBS teilt dem höchstprioren bereiten Thread, der an der Reihe ist, die CPU für die maximale Dauer seiner TICs zu.
►OS9000: Anzahl der TIC pro Thread =1 fest.

Thread-starten:

Zustandswechsel N => B (wenn möglich!)

►Starten des Hauptthreads einer Task: Vergabe einer eineindeutigen TaskID, Zuteilung der Betriebsmittel (OS9000➔Arbeitsspeicherzuteilung(ASP) aus dem vorhandenen freien Pool) ►Wenn noch nicht geladen, dann Laden des kompletten Programmcodes)
Beispiel: 4 Tasks: A,B,C,D
► ASP Grösse= ASP(A)+ASP(B)+ASP(C)+ASP(D) = 5► ASP(A)=ASP(C)=ASP(D) = 1, ASP(B)=2
► Startreihenfolge: Task A,B,C,D T=0 ► nach einem Zeitschritt Ende Task B und Task D, Task A und C laufen noch
► Nun Start Task D T=1► dann Versuch Start Task B failed!! No-Memory! Am Stück!!

T=0	A	B		C	D
T=1	A	D		C	
T=2	A	D	!	C	!

►Starten eines Threads: Vergabe einer einer eineindeutigen TaskID, Zuteilung der Betriebsmittel Stack (OS9000➔Arbeitsspeicherzuteilung(ASP) aus dem vorhandenen freien Pool) ►Ansprung des bereits geladenen Programmcodes

►OS9000: ThreadID wird vergeben, der Erzeuger erhält als Quittung für den erfolgreichen Start die TaskID/ThreadID der neuen Task/des neuen Threads durch das RBS mitgeteilt. Automatisch ist die/der startende Thread der Vater (parent) des gestarteten Threads (child).
►in OS9000 theoretisch max 65535 Threads im nicht N Zustand möglich ➔Begrenzung Speicher!!.

Thread-beenden
Zustandswechsel: B,W,R ►N

Thread wird in der Laufdatenverwaltung gelöscht, aller ASP und Betriebsmittel werden freigegeben ►Thread wieder im Zustand N

E 5.2.3 Thread- und Taskverwaltung

►**Anwender-Programmcode**, **Initialisierungsdaten (Konstante)** ►(OS9000: Prozessmodul Teil Objekt-Code, primary-module read-only, may be shared)

►**Anwender-Daten** (OS9000:Prozessmodul- Teil Daten-Area)

►**Laufzeitdaten und Laufzeitparameter** des RBS zur Beschreibung Task im ASP zusammengefasst im Taskkontrollblock/Threadkontrollblock,

- ► (OS9000 Beispiel siehe: process-header ►**Uprocess.h**)
- ► Prozesszustand (W,R,B)
- ► Art : OS9000: system/userstate-Task
- ► ProcessID, ThreadID
- ► Priorität = Dringlichkeit
- ► effektive Ausführungsrechte ► Prozessausführungsrechte # Eigentümer group/user z.B. (0,0) ► Ausführungsrechte auch z.B. Superuserrechte werden geerbt vom Starter (parent) (auf Taskebene)
- ► Arbeitsspeicherbedarf
- ► I/O- Verwaltungsplatz
- ► Aktivierungsangaben► aufgelaufene Zeit (virtuelle Prozesszeit).
- ►**Threadbeschreibung** pro Thread einmal:
  - ►Registersatz, Prozessorzustand ► User/ggf. System-Stackpointer (OS9000:Prozessmodul -Teil Daten)
- ►OS9000: ein Systemstack für SYSA-calls (Umschaltung der User-Stackpointer bei Threadwechsel)

Schutzmöglichkeiten von Task-Speicherbereichen je Task untereinander:

►**geschützt** durch integrierte und vom RBS kontrollierte Memory Management Unit (MMU) oder Memory Protection Unit (MPU) (►OS9000-Entwicklungskernel mit SSM-Modul aktiviert)

►**ungeschützt**, alle Tasks gemeinsam =>(OS9000: atomic-Kernel ► Laufzeitkernel schneller!!)

E 5.2.4 Task-/Threaderzeugung und -entfernung

► Das Erzeugen und Entfernen von Tasks (Hauptthreads) erfolgt durch andere Tasks/Threads entweder programmgesteuert oder durch Bedienerkommandos an der Konsole (OS9000: ein mshell-Kommando). Threads sind immer an eine Task gebunden und können deshalb nur innerhalb ihrer Task von anderen Threads ihrer Task gestartet werden.

Interaktives Starten von Tasks unter OS9000, mshell-Beschreibung (Auszug),

- **Die *mshell* startet nach Booten des Praktikumssystems auf der Console oder nach dem Login des Benutzers via Telnet als User Super, Passwort user und kreiert folgende *enviroment*- Variable**

  1. **PORT** Der Name des Terminals *z.B. /t1* ►nur via telnet
  2. **HOME** Ihr *home-directory* ►nur via telnet
  3. **SHELL** Die erste Task die durch Ihr *login* gstartet wurde
  4. **USER** login-Name ►nur via telnet
  5. **PROMPT und _SHELLPARAMS**
  6. **PWD** Aktuelles Verzeichnis

- Diese können mittels folgender Kommandos manipuliert werden: ***setenv,unsetenv,printenv***

z.B. $ setenv PATH .:/h0/CMDS:/d0/CMDS

- Einige eingebaute **shell-Kommandos:**

** <text> ►comment ; chd <path> ►Wechselt Data-Dir; chx <path> ►wechselt Execution-Dir*

ex <name> ►shell wird durch Prog. <name> ersetzt (gestartet); kill <procID> ►beendet Task procID

logout ►abmelden; profile <path> ►Führe in der Datei path enthaltene shell-commands aus

*w ►Warte auf Beendigung **eines** Kindprozesses; wait ►Warte auf Beendigung aller Kindprozesse*

setpr <procID> <new-priority> ►Ändere Priorität der Task procID

- **Starten von Tasks (process) vom mshell-Prompt aus:**

$ meinprogramm #40k ^130 </dd/TEST/myinput >/dd/TEST/myoutput >>/dd/TEST/myfehler

- *Es wird **meinprogramm** als Kindprozess gestartet und danach auf Beendigung gewartet.*
  - *►Zuerst sucht die shell das Moduldirectory nach dem **Modul meinprogramm** ab, ob es bereits geladen ist, wenn ja wird ggf. nur ein neuer Datenbereich für die Task angelegt und der geladene Code erneut gestartet wenn der Benutzer entsprechend den **Modulpermission**rechten dazu berechtigt ist.*
  - *►Wenn es nicht geladen ist wird über das **Filesystem** zuerst der execution-path (chx <path>) und dann entlang der PATH-Enviromentvariablen gesucht. Wenn die **Datei meinprogramm** gefunden wurde **und** Benutzer diese Datei **(Filepermissions)** ausführen darf, wird der Programmcode geladen. Der im **Modulkopf** enthaltene **Modulname** wird ins Moduldirectory eingetragen und es wird überprüft ob die im Modulkopf enthaltene **Modulpermission** es dem Benutzer erlaubt dieses Programm auszuführen.*

- ►*Zu allerletzt wird das Data-Dir durchsucht und falls meinprogramm gefunden wird, wird angenommen, dass meinprogramm ein Textfile ist welches shell Kommandos enthält.* ►*Start einer Kind-shell mit Übergabe der Datei und warten auf Beendigung.*

- ***der #*** *Modifier alloziert zusätzlichen Daten-Speicher zur Task, wenn weggelassen Größe aus Modulkopf*

- ***der*** *^ Modifier erlaubt es die Startpriorität zu verändern* ►*ohne Angabe hier 128*

- *< Input-, > Output-, >> Erroroutput-Umleitung von/in Datei* ►*default stdin,stdout,stderr ans Terminal... (>+ heißt anhängen an und >- überschreiben einer bereits existierende Datei)*

- *( ) wird benutzt um Kommandos zusammenzufassen: $(del *.backup; list stuff >p)&*

- **Mehrere Kommandos in einer Zeile werden, wenn durch ; getrennt, nacheinander ausgeführt.**

!Nach Beendigung des letzten Kommandos erscheint der shell-Prompt wieder!!

- **Mehrere Kommandos in einer Zeile werden, wenn durch & getrennt, parallel ausgeführt**

  & als letztes Zeichen in der Zeile bedeutet: Ausführung des/der Kommandos im Hintergrund
  ►shell-prompt erscheint wieder, weitere Eingaben möglich! (& Operator geht vor ; Operator!)
  !! Achtung die gestarteten Prozesse erben stdin,stdout und stderr vom Starter!!

- Werden die **Kommandos durch ! getrennt** so werden alle Prozesse parallel gestartet. Der Output des ersten Kommandos wird umgeleitet aus Input des zweiten Kommandos .... ►***Filter durch pipes***

- **Wildcards-Expansion (teilweise!!)**

  * steht für ein bzw. mehrere beliebige oder kein Zeichen
  ? genau für ein beliebiges Zeichen

- **Besonders:Eingabe von Control-c** ►Wird ein Prozess im Vordergrund gestartet **und** hat noch kein I/O auf das Terminal getätigt **dann und nur dann** kann man mittels ^c den Prozess nachträglich in den Hintergrund befördern und es erscheint der Kommandprompt wieder!!
  !!^c wird als Signalnr.3 immer an die Task geschickt die zuletzt I/O am Terminal getätigt hat!!

- **Besonders: Eingabe von Control-e** ►Wird ein Prozess im Vordergrund gestartet **und** hat noch kein I/O auf das Terminal getätigt **dann und nur dann** kann man mittels ^e den Prozess aborten und es erscheint der Kommandprompt wieder!!
  !!^e wird als Signalnr.2 immer an die Task geschickt die zuletzt I/O am Terminal getätigt hat!!

- **Jede Task die gestartet wird, erbt die Group.UserID des Erzeugungerprozesses.**
  ►Durch den login-Prozess wird ein User-Programm gestartet, dieses erhält die GroupUserID des Benutzers
  ►normalerweise die mshell
  ►Es gibt Programme die nur mit der SuperuserID (0,0) ausgeführt werden dürfen.

Programmgesteuertes Erzeugen einer Task (Hauptthread) unter OS9000 mittels Betriebssystemaufruf

- **Die Anbindung der SYSAs erfolgt bei OS9000 PPC über den einheitlichen *C-Bibilotheksaufruf _oscall()* , der einen Verweis auf einen Parameterblock als Aufrufparameter enthält. In diesem Parameterblock ist abgelegt welche OS9000 Funktion aufgerufen werden soll und ausserdem sind alle Parameter für diese aufzurufende Funktion im Parameterblock hinterlegt. In der C-Programmierumgebung wird diese Anbindung in den C-Libraries vor dem Benutzer verborgen.**
- **Kreieren eines neuen Prozesses von C aus geschieht generell mittels der Funktion *_os_exec()*►Kreieren eines nebenläufigen Prozesses (Subfunktion *_os_fork*)**

#include <const.h> #include <process.h> #include <types.h> #include <cglob.h>

error_code _os_exec (_os _fork, u_int32 priority, u_int32 pathcnt, void* modname, char ** argv, char **envp, u_int32 datasize, process_id * child_id, u_int32 type_lang, char orphan);

_os_fork	***Starte eine Task parallel zu dieser Task***
priority	***=0 gleiche Priorität wie Aufrufer, sonst Wert < 65535***
pathcnt	***Anzahl der übergebenen (kopierten) offenen Pfade ➔ i.d.R =3 stdin,stdout,stderr wie Aufrufer***
modname	***zeigt auf den Modulnamen des Programms das gestartet werden soll!! Wird das Modul als File geladen, so muss der Filename und der Modulname identisch sein (erzeugt der C-Compiler so)!!***
argv	***dienen zur Übergabe von Programm-Parameter an die neue Task (siehe Beispiel)***
envp	***verweist indirekt auf die Enviromentvariablen die für den neuen Prozess gelten sollen. Envp zeigt auf den Anfang einer Liste von Zeigern, die jeweils auf den Null-terminierten-Textstring einer Enviromentvariablen-zuweisung zeigen. Der letzte Zeiger in der Liste ist der NULL-Zeiger. ➔ Normal: ´_environ´ zeigt auf den eigenen Bereich***
datasize	***Größe von zusätzlich bereitzustellenden Speicherplatz für den Datenbereich der neuen Task ➔ Normal =0***
child_id	***Adresse einer Variablen die die procID der gerade erzeugten Task aufnimmt***
type_lang	***Modul-Type der Task : Normal: = mktypelang (MT_PROGRAM, ML_OBJECT)***
orphan	***=0 normales child, =1 ohne Vaterverweis, Kind ohne Eltern wenn ein normales child terminiert bleibt der Process-descriptor noch solange im Speicher bis der Vater dies bemerkt _os_wait() oder stirbt.***

- ***Beispiel:***

```
#include <const.h>  #include <process.h> #include <errno.h> #include <types.h> #include <cglob.h>
#include <stdlib.h> #include <const.h>

main() {
error_code      myerr;
u_int32         priority = 0;           /* gleiche Priorität wie der Aufrufer */
u_int32         paths =3;               /* die ersten drei offenen Pfade vererben*/
u_int32         edata = 0;              /* kein zusätzlicher Platz */
process_id      child_id;          /* enthält child-procID nach os_exec() Aufruf */
char            orphan =0;                    /* normales child!! */
u_int32         typelang;               /* Type des zu ladenden Moduls */

static char     * arguments [ ] = { "meinprogramm", "erstesArg", "zweitesArg", NULL};
            /* Aufruf via mshell waere: meinprogramm erstesArg zweitesArg */

typelang = mktypelang (MT_PROGRAM, ML_OBJECT);

if ( (myerr = _os_exec (_os_fork, priority, paths, arguments[0], arguments, _environ, edata,
                        &child_id, typelang, orphan) ) != SUCCESS) exit (myerr);
oder kürzer:

if ( (myerr = _os_exec (_os_fork, 0, 3, arguments[0], arguments, _environ, 0,
                        &child_id, mktypelang (MT_PROGRAM, ML_OBJECT), 0) ) != SUCCESS) exit (myerr);
```

- **Der Erzeuger kann auf Beendigung eines beliebigen Kindes mit *_os_wait()* warten:**

```
#include <const.h>  #include <process.h> #include <types.h> #include <myerr.h>
#include <stdlib.h>  #include <const.h>

error_code      myerr;
process_id      wer_war_es;
status_code     wie;

if ( (myerr =  _os_wait (&wer_war_es,&wie) ) !=SUCCESS ) exit (myerr);
```

Bei der Rückkehr ist in *wer_war_es* abgespeichert welcher child terminiert hat und in *wie* mit welchem Erfolg

- **Verketten statt Nebenläufig ➔ Ersetzen des aufrufenden Prozesses durch einen Nachfolger ➔ _os_chain (siehe C-Lib Ref.)**

Anmerkung zum Praktikum:

Umdruck: E5_Uos9errno_h_xxxx enthält die Fehlercodes, die jede aufgerufenen RBS-Funktion zurückgibt

Tasks (pocesses) und Threads (POSIX 1003.1c) unter OS9000: pthreads-Library

- **Bisher**: Eine Task (process) hat nur einen 'Ablauffaden' also einen Thread
  - Einen Stack (User), einen Registersatz, einen Programmcounter, eine Priorität, eine Signalinterceptroutine
  - einen globalen statischen Speicherbereich (Process Control Block) in dem Taskverwaltungsdaten w.z.B. Filehandles gehalten werden

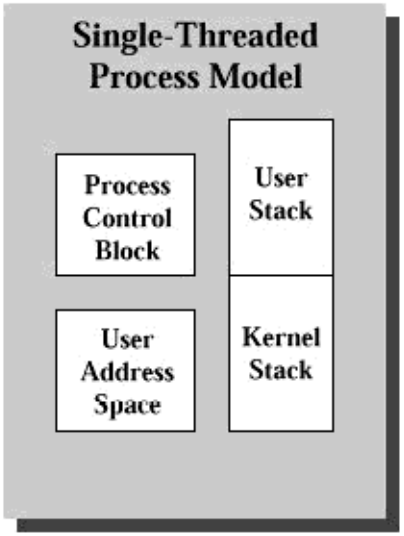
- **Wenn zwei oder mehrere Tasks an einer** Aufgabe zusammenarbeiten, dann müssen sie sich
  - i.d.R. zeitlich im Ablauf synchronisieren via Events, Semaphore oder Signals und
  - Daten austauschen via z.B. gemeinsamen Speicher (OS9000-Datenmodul)
  - -> Aufwand relativ groß, aber robust gegen Einzeltaskfehlverhalten, wenig Randbedingungen zu beachten

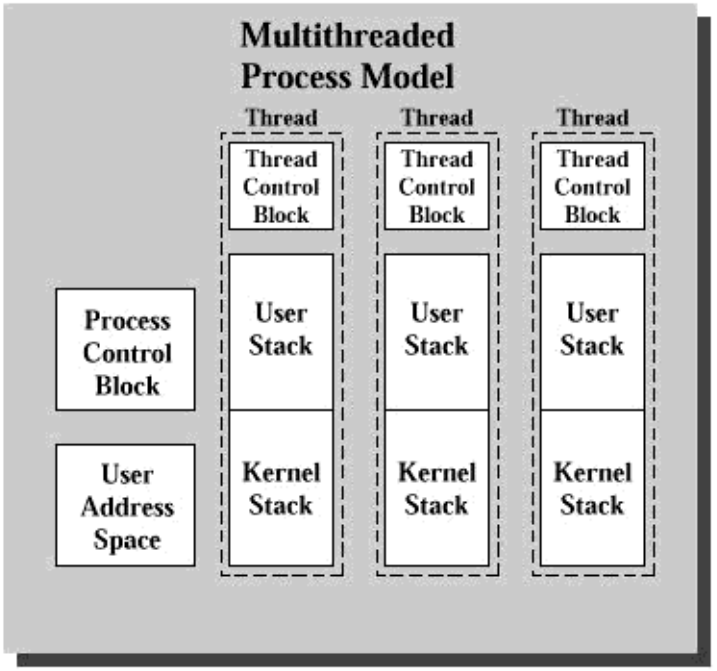
- **Idee**: Es gibt **in einer Task** (process) **mehrere Ablauffäden sog. Threads mit:**
  - alle Threads teilen sich einen **gemeinsamen** globalen statischen Speicherbereich in dem Taskverwaltungsdaten w.z.B. Filehandles gehalten werden
  - aber jeder Thread hat einen eigenen Stack, Registersatz, Programmcounter, Priorität, Signalinterceptroutine

- **Resultat:**
  - Datenaustausch zwischen Threads über gemeinsame programmglobale Variable der Task
  - Synchronisation zwischen Threads einer Tasks über Semaphore (Mutex, OS9000-Semaphore), Events (Condition Variable, OS9000-Events) und eingeschränkt über Signale
  - Nachteil: Wenn ein Thread 'abstürzt' und vom Betriebssystem zwangsbeendet wird, wird die gesamte Task beendet, d.h. alle Threads werden unkontrolliert beendet. -> Ungeeignet für Watchdoganwendungen, Einschränkungen bei der Programmierung zu beachten

Überblick über Taskmodelle mit einem oder mehrerem Threads

- In OS9000 ist eine adaptierte Portierung der sog. pthread-Library vorhanden, die den IEEE POSIX 1003.1c implementiert.





Threadbeispiel OS9000 pthreadtest.c

```
int iglobal = 10;
void * thread ( void* arg ) {
        char * pp;
        static int istatic=3;
        int i;
        pp = (char *) arg;
        i=1; i++; istatic++; iglobal++;
        printf ("%s i:%d istatic:%d iglobal:%d\n",pp,i,istatic,iglobal);
        pthread_exit( NULL ); }

/* Main ist der Hauptthread, endet er, dann wird alles beendet */
int main (int argc, char** argv) {
char * cb="Hallo thread ";
int i; static int istatic=2;
pthread_t p_id1,p_id2;
printf (" Mainstart \n"); i=1;
/* Starten eines Threads */
pthread_create( &p_id1, NULL , thread , (void *)cb ) ;
/* Starten eines Threads */
pthread_create( &p_id2, NULL , thread , (void *)cb ) ;

/* Warten auf Ende des ersten gestarteten Threads */
iglobal++;
pthread_join ( p_id1, NULL);

/* Warten auf Ende des zweiten gestarteten Threads */
pthread_join ( p_id2, NULL);
/* Ende */
printf ("Mainthread i:%d istatic:%d iglobal:%d\n",i,istatic,iglobal);
return 0; }
```

- **Aufruf von pthreadtest: $ pthreadtest**
  Mainstart
  Hallo thread i:2 istatic:4 iglobal:11
  Hallo thread i:2 istatic:5 iglobal:12
  Mainthread i:1 istatic:2 iglobal: 13
- **Analyse des Beispiels:**
  - **Alle Threadbibilotheksfunktionen beginnen mit** pthread_
  - **C-Funktionen werden zu einem Thread durch** pthread_create()-**Aufruf**
  - **Der Thread erbt die Priorität des Hauptthreads**
  - **Ein und dieselbe C-Funktion kann durch mehrere Threads gleichzeitig benutzt werden**
  - **funktionslokale Variable je Stack also je Thread; funktions static Variable für alle Threads nur einmal vorhanden; globale Variable einmal vorhanden und für alle Threads sichtbar**
  - **Erlaubtes Ende eines Threads nur durch** pthread_exit()
  - **Hier: Warten auf das Ende eines Threads mit Abholen des Ergebnisses durch** pthread_join()
  - **Jede Thread ID ist gleichwertig einer Task ID und systemweit zu sehen -> Signal**
  - **Es fehlt hier privater statischer Variablenplatz je Thread! Das unterstützt C nicht direkt!**

Threadeigenschaften OS9000

- Ein Thread hat eine ThreadID, Wertebereich 1-65535, und eine ParentID die angibt –wie die ParentID bei Tasks- welcher Thread diesen gestartet hat. Mittels **pthread_t pthread_self(void);**kann die eigene ThreadID ermittelt werden.

- ThreadIDs und TaskIDs stammen aus demselben Nummernpool: ►Es gibt keine zwei gleichen ThreadIDs bzw. TaskIDs

- Jede Task hat einen sog. **Hauptthread** , wird der beendet, dann werden alle Threads der Task beendet
- Die Hauptthread **ThreadID** ist identisch mit der **TaskID**

- Ein Thread hat seinen eigenen Registersatz und Stackbereich,
  - alle funktionslokalen Variablen sind threadlokal
  - alle funktionsstatischen Variablen sind nur einmal vorhanden aber nur innerhalb der Funktion sichtbar, alle Threads teilen sich die funktionsstatischen Variablen einer Funktion
  - alle globalen und statisch globalen Variablen sind nur einmal vorhanden und alle Threads teilen sich diese Variablen
  - **Jeder Threadzugriff auf eine globale oder statische Variable muss** -z.B. via *pthread_mutex_xxx()*- **sychronisiert** werden!! um Inkonsistenz der Variableninhalte zu vermeiden

- Ein Thread hat seine eigene veränderbare **Priorität**

- Ein Thread kann von jedem Thread **Signale** empfangen

- Ein Thread kann **Events** anlegen, löschen ,auf Events warten und signalisieren

- Ein Thread hat voreingestellt eine Signalempfangsroutine zur Behandlung von threadeigenen Signalen
  - Ein Thread darf **nicht _*os_intercept()*-**Aufruf zur Installation einer eigenen Signalintercept-Funktion verwenden; der Hauptthread schon.
  - Zur Installation einer Signalintercept-Funktion kann die Funktion **intercept()** oder **signal()** verwendet werden. Achtung ! es darf nicht alles in der Signalintercept-Funktion verwendet werden!

- **die Beendigung eines Threads durch einen Fehler mit Zwangsabbruch des Threads führt zum Zwangsabbruch der gesamten Task inklusive aller anderen in dieser Task noch laufenden Threads!**

- Der Hauptthread kann 'normal' z.B. mit exit() oder return() oder "}" beendet werden, alle anderen Threads einer Task müssen sich mit pthread_exit() beenden**.**

- Threads laufen entweder “detached“ oder “normal“
  - „detached“ Threads sind Orphans, deren **ParentID** Null ist und auf deren Beendigung nicht gewartet werden kann
  - „normal“ Threads beitzen eine von Null verschiedene **ParentID** ; auf deren Beendigung kann mittels pthread_join() gewartet werden. Der Rückgabewert eines solchen Threads geht bei Beendigung des Threads nicht verloren und wird bis zu Abholung aufbewahrt oder bis zur Beendigung des Hauptthreads.

- !!!! Threads gibt es zunächst nur für **user –state** Programme!!!

Threaderzeugung, -beendigung und –steuerung

- Vorbereitung: Es muss darauf geachtet werden, dass die Thread-sicheren Bibliotheken eingebunden werden (Entwicklungsumgebung unter Eigenschaften einer Komponennte ->Source->Multithreading)
- Benötigte Prototypen-Headerdatei: **#include <pthread.h>**
- Alle folgenden Funktionsaufrufe funktionieren nur für Threads innerhalb einer Task **und dürfen nicht innerhalb eines durch pthread-Mutex geschützten Bereichs aufgerufen werden**.

- **Threadeigenschaften vor Start einstellen**, dazu wird z.B. eine Variable **attr** vom Typ *pthread_attr_t* definiert und mit Werten belegt : ***pthread_attr_t attr*; /* Beschreibungskontainer */**

  - Mittels Aufruf der Funktion **pthread_attr_init(pthread_attr_t *attr);** wird die Stuktur die sich hinter der Variablen attr verbirgt auf default-Werte initalisiert:

1.	**Stack Size**	**PTHREAD_STACK_MIN**
2.	**Stack Address**	**NULL (system allocated stack)**
3.	**Detach State**	**PTHREAD_CREATE_JOINABLE**
4.	**Priority 0**	**(priority of creator)**
5.	**Initialization Function**	**NULL (none)**

- Thread wird mittels **pthread_attr_setdetachstate(pthread_attr_t *attr, PTHREAD_CREATE_xxxx)**;Aufruf entweder
  - **PTHREAD_CREATE_DETACHED**,
    - das heisst er ist ein Orphan (Weise) auf dessen kein anderer Thread warten kann und bei dessen Beendigung alle belegter Threadplatz automatisch freigegeben werden.
  - oder **PTHREAD_CREATE_JOINABLE** (**default**) erzeugt
    - auf die Beendigung kann mittels **pthread_join()** Aufruf gewartet werden. **Wird ein solcher thread beendet so muss mittels *pthread_join()* der thread-Rückgabewert abgeholt werden**, sonst 'hängt' der thread als 'ghost' in der process-Tabelle.
- Die Threadstartpriorität kann mittels ***_pthread_attr_setpriority(pthread_attr_t *attr,u_int32 priority*)*** eingestellt werden. Der Wert 0 besagt, dass dieselbe Priorität wie der Erzeuger eingestellt werden soll.

- **<u>Threaderzeugung:</u>** Die Startpriorität des Threads ist default dieselbe wie des Erzeugers

```
pthread_create (
        pthread_t *thread,                  /*Out: ThreadID*/
        const pthread_attr_t *attr,          /*In: Threadstarteigenschaften NULL : default Einstellungen siehe
                                             pthread_attr_init()
    void *(*start_routine) (void *),        /*In: C-Funktion deren Code als Thread laufen soll, ein Parameter möglich
                                            Typ: generischer Zeiger /*
        void *arg);                         /* Verweis auf den zu übergebenen Parameter, Typ-Casting erforderlich,
                                            Achtung! bei Übergabe von Verweisen auf statische oder globale Variable,
                                            die existieren für alle Threads nur einmal! */
```

- **Taskbeendigung:**

  - Selbstbeendigung des Threads mittels: **void pthread_exit(void *value_ptr); value_ptr** verweist auf einen Wert, der als Resultat zurückgegeben werden kann. ***Vorsicht: Der Verweis muss auf einen Speicherort verweisen, den es nach Beendigung des Threads noch gibt***.

  - Zwangsbeendigung eines anderen Threads: int **pthread_kill(pthread_t thread, int sig);** mit **sig=0** oder **int pthread_cancel(pthread_t thread); thread** ist die ThreadID des abzubrechenden Threads. Der abzubrechende Thread wird im Falle cancel falls er im Status **PTHREAD_CANCEL_ENABLE** (default) ist, beendet, sonst wird vom System der Abbruch verzögert bis der Thread wieder in diesem Zustand ist. Der Abbruchaufruf kommt immer sofort zum Aufrufer zurück. !!Achtung der thread ist ‚wenn er 'joinable' ist, solange ein 'ghost'-process, bis mittels **pthread_join()** Aufruf der 'ghost' beendet wird!!!

- **Tasksteuerung:**

  - Das Warten auf das Ende eines anderen Threads, der im “normal” Modus läuft –also nicht „detached“ ist- ist durch: **int pthread_join(pthread_t thread, void **value_ptr);** möglich.
    - **thread** ist die ThreadID des Threads auf dessen Beendigung gewartet werden soll;
    - **value_ptr** ist die Adresse einer Variablen, die in der Lage ist einen Verweis auf das Ergebnis des beendigten Threads aufzunehmen.

  - Eine Task kann sich gegen das Fremdabbrechen schützen: **int pthread_setcancelstate(int state, int *oldstate); state** kann sein**: PTHREAD_CANCEL_ENABLE** or **PTHREAD_CANCEL_DISABLE**.

- Die Art der Fremd-Abbrechbarkeit kann gesondert mittels **int pthread_setcanceltype(int type, int *oldtype);** eingestellt werden. Die Art wird über **type** eingestellt und kann entweder **PTHREAD_CANCEL_DEFERRED** oder **PTHREAD_CANCEL_ASYNCHRONOUS** sein.
  - **PTHREAD_CANCEL_DEFERRED** bedeutet der Abbruchwunsch wird vom System solange verzögert bis der abzubrechende Thread einen Aufruf von **void pthread_testcancel(void);** tätigt.
  - **PTHREAD_CANCEL_ASYNCHRONOUS** (default) bedeutet ein Abbruchwunsch wird sofort ausgeführt .

- Mittels **int *_pthread_suspend(pthread_t thread, unsigned int *count);*** kann ein Thread **thread** angehalten werden und mittels ***int _pthread_resume(pthread_t thread, int *status);*** wieder geweckt werden. Änlich einem Linkcount werden die Anzahl der Suspendaufrufe gezählt und erst ein Wecken ausgeführt wenn eine korrespondierende Anzahl von Resumeaufrufen erfolgt ist.

- Mittels **int *_pthread_setpr(pthread_t thread, u_int32 priority);*** kann zur Laufzeit eines Threads die Priorität verändert werden

OS9000-Shellkommando procs (Display Processes)

- **procs dient zur Anzeige der laufenden Tasks im System. Ohne weitere Parameter zeigt procs die Tasks des eingeloggten Benutzers an. procs -e zeigt alle im System befindlichen Tasks an:**

$ procs -e

Id	Pld	Thd	Grp.Usr	Prior	MemSiz	Sig	S	CPU Time	Age	Module & I/O
2	5	1	0.0	128	38.00k	0	w	0.02	???	mshell <>>>term
3	0	1	0.0	128	103.50k	0	s	0.05	???	inetd <>>>nil
4	0	1	0.0	128	16.25k	0	e	0.02	???	spf_rx
5	0	1	0.0	128	6.25k	0	w	0.01	???	sysgo <>>>term
6	0	1	0.0	128	6.50k	0	s	0.00	???	ndpio <>>>nil
7	9	1	0.0	128	66.50k	0	*	0.08	0:00	procs <>>>term
8	0	1	0.0	128	26.25k	0	s	0.02	???	spfndpd <>>>nil
9	2	1	0.0	128	38.00k	0	w	0.13	???	mshell <>>>term

Id	Process ID
PId	Parent process ID
Thd	Thread count
Grp.usr	Owner of the process (group and user)
Prior	Initial priority of the process
MemSiz	Amount of memory the process is using
Sig	Last pending signal value for the process/exit status for dead process
CPU Time	Amount of CPU time the process has used
Age	Elapsed time since the process started

S	Process status: * Currently executing w Waiting s Sleeping a Active e Waiting on event m Waiting for an mbuf z Suspended process
Module & I/O	Process name and standard I/O paths: < Standard input > Standard output >> Standard error output If several of the paths point to the same pathlist, the identifiers for the paths are merged.

- **procs -a zeigt weitere Informationen an:**

  $ procs –a

Id	Pld	Thd	Aging	F$calls	I$calls	Last	Read	Written	Module & I/O
2	5	1	128	81	18	F_WAIT	0	0	mshell <>>>term
3	0	1	128	147	45	F_SLEEP	0	0	inetd <>>>nil
4	0	1	128	0	0	???	0	0	spf_rx
5	0	1	128	20	0	???	0	0	sysgo <>>>term
6	0	1	128	12	2	F_SLEEP	0	0	ndpio <>>>nil
7	9	1	128	92	23	F_GPRDSC	0	457	procs <>>>term
8	0	1	128	84	14	I_SETSTAT	0	0	spfndpd <>>>nil
9	2	1	128	131	184	F_WAIT	51	68	mshell <>>>term

Aging	Age of the process based on the initial priority and how long it has waited for processing
F$calls	Number of service request calls made
I$calls	Number of I/O requests made
Last	Last system call made
Read	Number of bytes read
Written	Number of bytes written

- **procs –x zeigt detailliertere Informationen an**

- **procs –t zeigt Threadinformationen zu einem Prozess an**

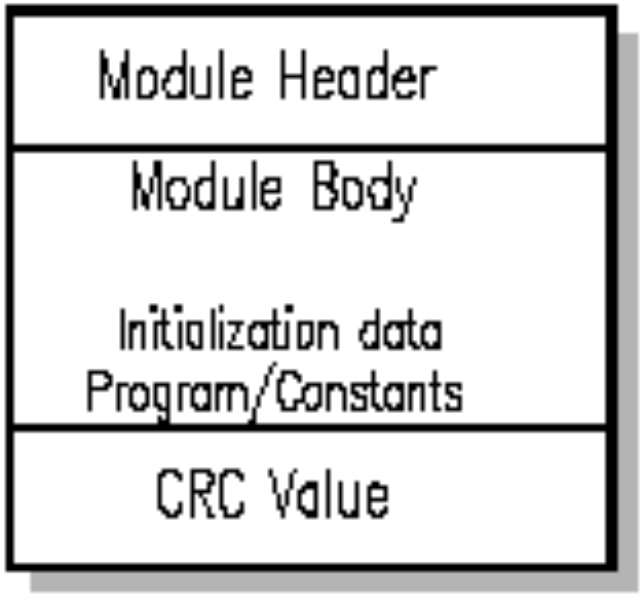
```
$ procs –t 7
 Id  Pld Thd Grp.Usr  Prior  MemSiz Sig S    CPU Time   Age Module & I/O
  7   9   2     0.0     128   17.75k   0 a        22.23  0:01 ptest <>>>term
 10   7   -     0.0     128       -    0 a        22.25  0:01
```

- **kill <TaskID> terminiert sofort die Task TaskID**

E 5.2.5 OS9000 Modulearchitektur

- Bei OS9000 ist das sog. **Modul** die übergeordnete Einheit die im Arbeitsspeicher (ASP) vom Betriebssystem verwaltet wird:

- Modulaufbau:



- CRC-Wertberechnung: 24-bit CRC über das erste Byte bis zum letzten Byte (ohne CRC) des Moduls
  ►**Module werden nur dann als Module akzeptiert, wenn ein gültiger Modulkopf vorliegt und! der CRC i.O. ist !**
- Alles sind Module -selbst der OS9000-Kernel- :
  - Programme, Unterprogramme,
  - Datenbereiche
  - Devicetreiber, Devicedescriptoren, Filemanager
  - Systemprogramm, Spezialmodule CSD
  - Libraries (Trapmodule)

- Ausführbare-Module müssen (bis auf wenige Ausnahmen) sein: **reentrant und position-independent**
- Alle im ASP befindlichen Module werden im ASP im sog. Modulverzeichnis verwaltet.
- **Modulkopf** (offset Angaben vom Modulstart ab gerechnet):

```
0x0000h:   u_int16  m_sync,          /* sync bytes ►PDT MPC 555-System: 0xF00D  */
0x0002h:   m_sysrev;             /* system revision check value */
0x0004h:   u_int32 m_size;       /* module size ►Alles!! incl CRC Body*/
0x0008h:   owner_id m_owner;     /* group/user ID */
0x000Ch:   u_int32 m_name;           /* offset to module name ►PDT: Offsetwert 0x0058h*/
0x0010h:   u_int16 m_access;     /* access permissions */
```

  - Die Zugriffsmöglichkeiten auf ein Modul sind an die *Access -Permission* Rechte gebunden.
    m_owner (group/userID) gibt an wem das Modul gehört.
    m_access legt für MP_OWNER_*, MP_GROUP_*,MP_WORLD_* jeweils die EXEC/READ/WRITE Rechte fest.
  - Bei Zugriff auf ein Modul (anlinken,ausführen..) wird überprüft ob der Benutzer, identifiziert durch seine group/userID, berechtigt ist oder nicht.

0x0012h: u_int16 m_tylan; /* **module type :**
MT_ANY 0 = Not used (wildcard value in system calls)
MT_PROGRAM 1 = Program module
MT_SUBROUT 2 = Subroutine module
MT_MULTI 3 = Multi-module (reserved for future use)
MT_DATA 4 = Data module
MT_CDBDATA 5 = Configuration Data Block data module
MT_TRAPLIB 11 = User trap library
MT_SYSTEM 12 = System module
MT_FILEMAN 13 = File manager module
MT_DEVDRVR 14 = Physical device driver
MT_DEVDESC 15 = Device descriptor module
and language:
ML_ANY 0 = Unspecified language
(wildcard in system calls)
ML_OBJECT 1 = Machine language
ML_ICODE 2 = Basic I-code (reserved for future use)
ML_PCODE 3 = Pascal P-code (reserved for future use)
ML_CCODE 4 = C I-code (reserved for future use)
ML_CBLCODE 5 = Cobol I-code (reserved for future use)
ML_FRTNCODE 6 = Fortran

```
0x0014h:  u_int16 m_attrev;     /* module attributes (first byte) and revision (second byte)
                                Bit 7 The module is re-entrant
                                (sharable by multiple tasks).
                                Bit 6 The module is sticky. A sticky module is not removed from memory until its link count
                                becomes -1 or memory is required for another use.
                                Bit 5 The module is a system-state module.
                                Wenn zwei Module gleichen Namens im Speicher sind, so wird nur das Modul
                                verwendet, dass die höhere Revisionsnummer besitzt. */
0x0016h:  u_int16 m_edit;       /* module edition number ►semantic User-definable */
0x0018h:  u_int32 m_needs,      /* module hardware requirements flags ►not used!*/
          /* (reserved) */
0x001Ch:  m_share,              /* offset of shared data in statics */
0x0020h:  m_symbol,             /* offset to symbol table ►reserved*/
0x0024h:  m_exec,               /* offset to execution entry point ►Bedeutung hängt vom Modultyp ab
                                                                    ►Datamodul Offset zum ersten Datenbyte*/
0x0028h:  m_excpt,              /* offset to exception entry point*/
0x002Ch:  m_data,               /* data storage requirement ►exec-Modul*/
0x0030h:  m_stack,              /* stack size ►exec-Modul */
0x0034h:  m_idata,              /* offset to initialized data */
0x0038h:  m_idref,              /* offset to data reference lists */
0x003Ch:  m_init,               /* offset to initialization routine*/
0x0040h:  m_term,               /* offset to termination routine */
0x0044h:  m_dbias,              /* data area pointer bias*/
0x0048h:  m_cbias;              /* code area pointer bias */
0x004Ch:  u_int16 m_ident;      /* linkage locale identifier */
0x004Eh:  char m_spare[8];      /* reserved */
0x0056h:  u_int16 m_parity;     /* header parity One's complement of the exclusive-OR of the previous header words. OS-9
                                      uses this field to check module integrity.*/
```

OS9-Shell-Commands für Modulinfo und -verwaltung

- **mfree -e** **Display Free System Memory**
- **mdir [<opts>] [<modname>]** **Display Modul Directory (➔ im Speicher)**

Beispiel : $ mdir -e

Current Module Directory

Addr	Size	Owner	Perm	Type	Revs	Ed	# Lnk	Module name
fff58f28	4016	1.0	0555	Prog	c001	5	1	activ
01050000	6320	1.0	0555	Prog	c001	37	1	attr
ffe0aba0	14776	1.0	0555	Prog	c001	208	1	bootptest
fff09368	6200	0.0	0555	Sys	a000	18	1	bootsys
fff5b1a0	3256	0.0	0555	Prog	c001	10	1	break
fff041b0	1088	0.0	0555	Data	8000	1	1	cnfgdata
fff045f0	4408	0.0	0555	Sys	a000	16	1	cnfgfunc
fff56340	11240	1.0	0555	Prog	c001	52	1	copy
01014420	72184	1.0	0555	Subr	c000	19	4	csl
fff5be58	4456	1.0	0555	Prog	c001	23	1	date
010587a0	3840	1.0	0555	Prog	8001	7	1	datmod2file
0105a0a0	192	0.0	0333	Data	8000	1	1	datstartup
fff87a90	16232	1.0	0555	Prog	c001	33	1	dcheck

►Es gibt voreingestellt zwei Pfade : Datemodulpfad und Executionmodulpfad
►Bei versuchtem Programmstart von dem Shell-Prompt aus wird in folgender Reihenfolge gesucht:
1. Moduldirectory im Speicher 2. Executionmodulpfad
➔Beim programmgesteuerten Laden von Datenmodulen:
1. Moduldirectory im Speicher 2. Datenmodulpfad

►Das OS9- Moduldirectory kann Submoduldirectories beinhalten ➔nicht in PDT

- **pd** — **Print Working Directory ►aktuelles Datenmoduldirectory**
- **pd -x** — **Print Working Directory ►aktuelles Executionmoduldirectory**
- **chd [<path>]** — **Change Current Data Directory**
- **chx <path>** — **Change Current Execution Directory**
- **load [<opts>] {<file>}** — **Load Modul from File into Memeory**
  - **►Läd die Datei <file> in den Speicher. <file> wird zunächst im Executiondirectory gesucht und dann entlag der Enviromentvariablen *PATH***
  - **►*-d* Option läd aus dem Datenmoduldirectory.**
  - **►In Abhängigkeit vom *revision*-Level wird ein gleichnamiges Modul im Speicher ggf. überschrieben**
  - **►Beim Laden werden die *modul-permissions* überprüft owner/group/public**
- **unlink {<modulname>]** — **Unlink Memory Modul im Arbeitsspeicher**
  - **►wenn der Link-Count = 0 (und nicht in *bootlist* or *sticky*) dann wird das betreffende Modul gelöscht**

Aufgabe: Ein bereits geladenes!! Modul soll erneut vom Massenspeicher geladen werden

►mittels mehrfach Aufrufs von *unlink* <*modulname*> zuerst völlig entladen!! ► Link-count auf „0“

OS9-Geräte- und Dateisystem

- **OS9 hat ein hierarchisches Dateisystem.** ►Directory/Subdirectory -Struktur beliebig verschachtelt.
- **OS9-Dateinamen unterscheiden nicht Groß und Kleinschreibung!! ➔!! Aber mit abgespeichert!!**

►Dateinamen können bis zu 43 Zeichen lang sein. Gültige Zeichen:A-Z,a-z,0-9,_, .,$
►*CMDS* und *cmds* sind ein und dasselbe!! Keine zwei Dateien!! *Konvention: Dirnames in Großb.*
►Als Programmparameter eines shell-Kommandos findet jedoch keine Konvertierung statt!!

- **Zu jeder Datei werden Namen, Dateianfangsposition , Dateilänge, Datum der letzten Modifikation abgespeichert**
- **OS9 unterstützt Zugriffsschutzmechanismus für Dateien**
  - Jeder Benutzer hat eine eigene ***user/group-ID*** , die durch den Login-Prozess (*password-Datei)* festgelegt ist. ➔Group/User 0,0 ist der Super-User und der darf alles.
  - Jede Datei gehört zunächst dem Ersteller (owners *user/groupID) =owner.*. Änderung der Rechte nur durch *super-user* oder *owner (oder alle mit der gleichen groupID)* mittels *attr Kommando* möglich.
  - Jede Datei enthält ein Feld zur Beschreibung der Zugriffsrechte für den *owner* und *public*

$ dir -e

Directory of . 13:32:32

Owner	*Last modified*	*Attributes*	*Block*	*Bytecount*	*Name*
0.0	*03/10/06 0000*	*d----swr-swr-swr*	*7*	*128*	*CMDS*
0.0	*03/10/06 0000*	*------wr------wr*	*A*	*70*	*STARTUP*
0.0	*03/10/06 0252*	*d----swr-swr-swr*	*5*	*192*	*SYS*
0.0	*03/10/06 1332*	*----------wr*	*C*	*41*	*password*

Bitgruppen: do---public-group-owner
Position 1. d=directory 2. o single user file 6/10/14 s=search permission if directory/ e= executable
7/11/15 w=write allowed 8/12/16 r= read allowed

- **Jedes benutzbare Geräte hat einen systemweiten eineindeutigen Namen.**

z.B. serielle Devices: term (primary system device!!) t1 t2...
default disk drive: /dd RAM-Disk: /r0 Pipe-device: / pipe Null-device /nil

- **Der absolute Zugriffspfad** (*pathname*) einer Datei setzt sich aus dem Gerätenamen, dem Directorypfad und dem Dateinamen selbst zusammen. Der Gerätenamen wird vorangestellt:

$ list /dd/CMDS/BOOTOBJS/test.c

- relative Zugriffspfadangaben erfolgen **ohne** vorangestelltem **/ und** beziehen sich immer auf die aktuelle Pfadposition die durch die vorangegangenen *chd*-Kommandos eingestellt wurde►*pd* zeigt diese Position

```
$ pd
/r0/SYS
$chd  ..
$pd
/r0
$chd SYS
```

E 5.2.5 OS9000 Systemstart

► Nach dem Einschalten und Hochfahren des OS9000 Kernels, dessen Einsprungpunkt z.B. im sog. Resetvektor der CPU hinterlegt werden kann, wird am Ende der Initialisierung des OS9000 eine Befehlssequenz, die im sog Init-Modul hinterlegt ist, abgearbeitet. ► I.d.R wird dadurch mindestens eine Urtask (➔OS9000 z.B. sysgo/sysgo01) des RBS gestartet.

►Die Urtask wird i.d.R. nie beendet.

►Die Urtask im Praktikum startet eine Kommandoshell mshell für den Benutzter am Terminal und wartet dann auf Beendigung dieser mshell um sie dann bei Beendigung durch den Benutzer gleich wieder zu starten (►Endlosschleife mit sleep). Somit wird im Praktikum sysgo nie beendet.

►Die Urtask hier: Sysgo ist ein vom Benutzer frei gestaltbares C-Programm ►Somit kann man statt der mshell auch andere Programme automatisch nach dem booten starten. (Multiuser/multitasking-Mode)
►Anmerkung zum Praktikum:
Umdruck: E5_Usysglob_xxxx listet die systemweit verfügbare Einstellungen und Parameter von OS9000 und die Zugriffsmethode auf selbige.

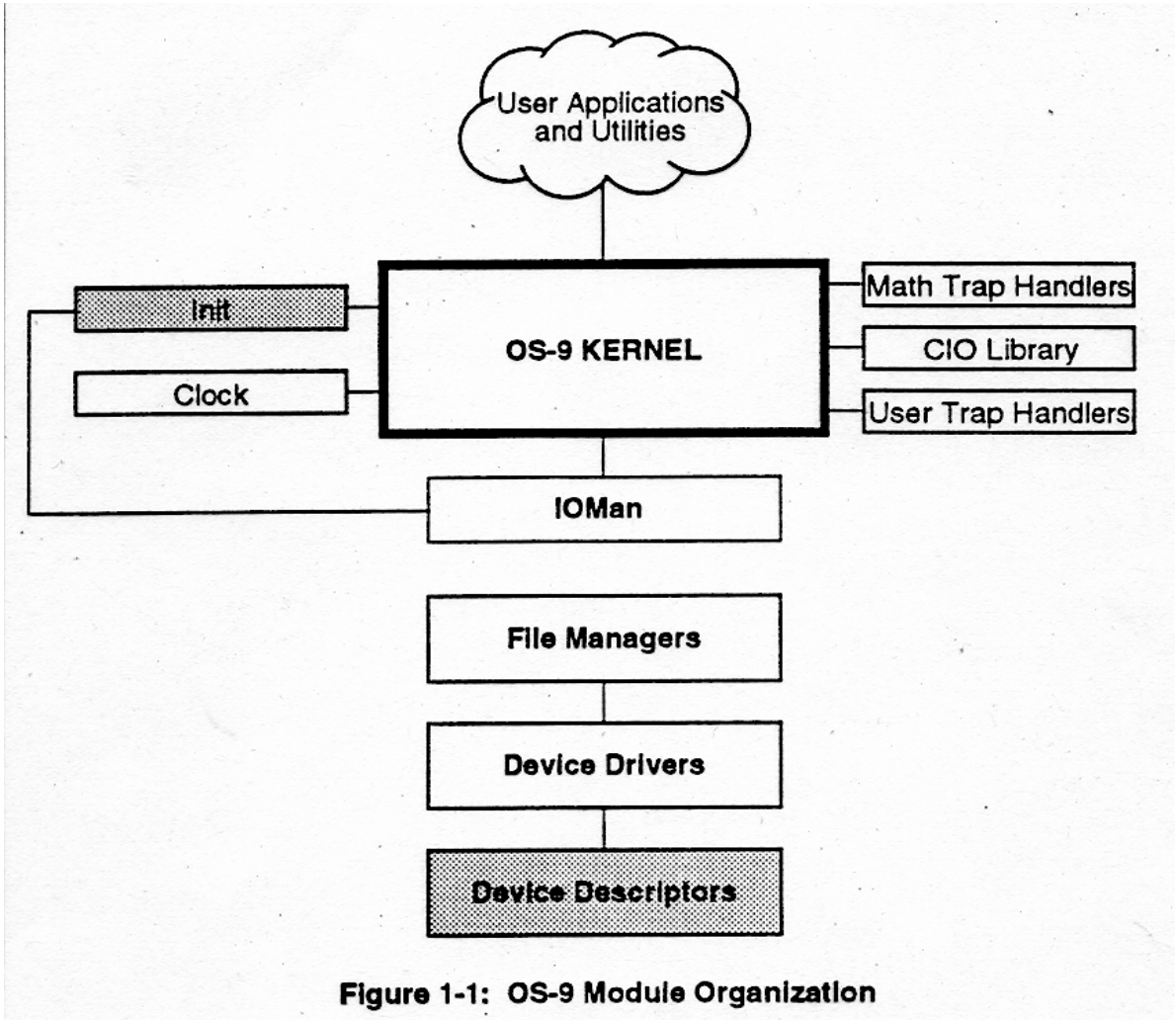


Figure 1-1: OS-9 Module Organization

E 5.3 Tasksynchronisation

Taskumschaltung durch aktive Steuerung der Prioritäten der Tasks zur Laufzeit

►Priorität = Dringlichkeit oder Vorrang ►Ein Prozess mit höherer Priorität verdrängt alle Prozesse mit niedriger Priorität
►Der Systemverwalter ordnet jedem Prozess zum Startzeitpunkt eine Priorität zu.
►Prioritäten können zu Laufzeit jederzeit taskgesteuert oder durch RBS geändert (OS9000 aging) werden.
- ► via mshell intrinsic command setpr <taskid> <prio>
- ► Ultra-C *_os_setpr()*

►hier: OS9000-Prios durch Task-Initiative änderbar (interaktiv,taskgesteuert) ►Prioritäten über Maxage sind statisch sonst Aging
►z.B. Überwachungstasks erhalten höhere Priorität als Protokolltasks

Freiwillige vorzeitige Rückgabe via Taskscheduler:

► Eine (Timesharing-)Task kann freiwillig die CPU-Zeit an das RBS vor Ablauf der Zeitscheibe und einer etwaigen Unterbrechung zurückgeben durch Aufruf des SCHEDULERS ►OS9000: _os_sleep(2) mindestens 1 TIC (=2 Timeslices) schlafen!!.

Sperren des Taskwechsels (task-locking)

► Es gibt im RBS i.d.R. die Möglichkeit unabhängig der eigenen Priorität einen ununterbrechbaren Task-Programmabschnitt zu erzeugen. ►Zugriff auf shared memory-Bereiche ►Gefährlich wg. Realzeitbdg.

SYSA: OS9000 so nicht vorhanden ►System-mode Programme können Interrupts sperren ►temporäres Erhöhen der Taskprio auf Maximum
➔!!aber bei Verwendung in Multiprozessorsystemen um Zugriff auf gemeinsamen Speicher zu sychronisieren ungeeignet
➔locked nur auf einem Prozessor/Prozessorkern alle anderen aus!!

Vertagung und Fortsetzung mittels Events

► Vertagung bis ein Ereignis (event) eintrifft. Anwendung:
- ► warten auf Nachricht von anderer Task oder externem Gerät
- ► warten auf Eventerzeugung durch Interrupt von der Prozessperipherie
- ► warten auf Timerablauf (Alarme,Sleeps)

► Task im Wartezustand wird in die wait-for-Event-Queue eingereiht. Die Queue ist sortiert nach dem FCFS-Prinzip nicht nach Prioritätsgesichtspunkten.

► Sie wird dort wieder ausgetragen durch Eintritt des Events ►OS9000: ... oder durch Auftreten eines signals für die wartende Task

OS9000-Events

- Jedes Event muss vor Verwendung vom Benutzer explizit unter Vergabe eines systemweit eindeutigen **Eventnames** angelegt werden (alle Events sind global bekannt).
- Ein **Event** ist ein vorzeichenbehafteter 32-bit **Zahlenwert.**
- Jeder ist berechtigt jedes existierende Event zu benutzen oder eins zu erzeugen.
- **Eventbezogen kann auf das Auftreten eines bestimmten Zahlenwertes oder auch auf das Erreichen eines vorgebbaren Wertebereichs des Zahlenwertes gewartet werden (➔event-queue).**
- **Die programmgesteuerte Veränderung des Datenwertes eines Events entspricht logisch dem Setzen des Events ►Der veränderte Datenwert wird vom OS9 mit den Wartebedingungen von Tasks auf dieses Event verglichen und diese ggf. rescheduled.**
- Alle angelegten Events werden systemweit in einer doppelt verketteten Liste vom Betriebssystem verwaltet. Ein Listenelement ist im wesentlichen wie folgt aufgebaut:

Event ID (32-bit)
Länge des Eventnamens(16 bit) **Zeiger auf Eventname**
Linkcount (links)
Zugriffsrechte (MP_OWNER_READ | MP_OWNER_WRITE |MP_GROUP_READ | MP_GROUP_WRITE |MP_WORLD_READ | MP_WORLD_WRITE **)**
OwnerID
Waitinkrement (Winc 16-bit)
Signalisierungsinkrement (Sinc 16bit)
Zahlenwert (32-bit)
Pointer to next event in event-list (next)
Pointer to previous event in event-list (prev)

- **Ein Event wird mittels *_os_ev_creat()* erzeugt**. Damit auch andere Tasks dieses Event verwenden können vergibt der Erzeuger einen systemweit eindeutigen **Eventnamen** unter dem es u.a. auch auffindbar ist.

►Aus Geschwindigkeitsgründen wird neben dem Namen noch eine systemweit eindeutige **EventID** geführt, die es dem Betriebssystem gestattet ohne Stringvergleich auf das Event zuzugreifen.

►Alle weiteren Dienste zur Eventnutzung benutzen daher als Übergabeparameter die **EventID** anstelle des Eventnamens.

►Beim Anlegen des Events initialisiert der Benutzer den **Zahlenwert des Events**

►**Winc** ist das sog. Waitinkrement. Wird eine Task, die auf einen Eventzustand wartete, geweckt so wird Winc auf den Zahlenwert des Events addiert.

►**Sinc** ist das sog. Signalisierungsinkrement. ´Setzt´ eine Task mittels SYSA das Event so wird der Wert Sinc auf den Zahlenwert des Events addiert. Anschließend wird, falls gewünscht, mit dem neuen Wert die Event-Warteliste durchgesehen ob eine Task zu wecken ist.

►Der Linkcount wird vom OS9 verwaltet und angelegt. Er gibt an wieviele Tasks dieses Event aktuell benutzen. Nur Events mit Linkcount = 0 können wieder aus dem System entfernt werden. Beim Erzeugen des **Events** wird der **Linkcount** auf eins gesetzt.

►Die beiden Pointer **next** und **prev** dienen zu Verkettung der Events zu einer Liste und werden von OS9 angelegt und verwaltet.

►Das **Event** wird unter der Benutzerkennung des erzeugenden Programms angelegt. Damit jeder mit dem Event arbeiten kann, müssen die Zugriffsrechte beim Erzeugen des Events entsprechend gesetzt werden.

- ***Beispiel Eventerzeugung**:*

```
#include <events.h>
#include <types.h>          ►Includes für alle Event-Calls
#include <const.h>
#include <memory.h>

int32     winc,sinc,value;
error_code   myerr;
event_id    id_pdv_event;
static char event_name [] = "PDV_event";

winc = 0; /* wait-inkrement soll nicht sein */
sinc = 1; /* jedes ´setzen´ erhöht den Wert um eins*/
value = 0; /* Startwert des Zahlenwertes */
u_in32 perm = MP_OWNER_READ  | MP_OWNER_WRITE |MP_GROUP_READ |
                          MP_GROUP_WRITE |MP_WORLD_READ | MP_WORLD_WRITE ;
```

myerr = _os_ev_creat (winc, sinc, perm , &id_pdv_event, &event_name[0], value, MEM_ANY);
*If (myerr != SUCCESS) { /*Fehlerbehandlung */ }*

► *bei Rückkehr wird in id_pdv_event die EventID abgespeichert ►Weitere SYSAs nutzen diese ID*
MEM_ANY gibt OS9 die Möglichkeit dieses Event in einem der möglichen Speicherbereiche anzulegen

- **Anlinken an ein existierendes Event** mittels Namen durch ***_os_ev_link ()*** ►***Linkcount*** *um eins erhöht*:

```
event_id   id_pdv_event;                    /*►enthält nach Aufruf die EventID */
static char event_name [] = "PDV_event";
error_code  _os_ev_link (event_name, & id_pdv_event);
```

- **Event setzten:** ***_os_ev_set()*** **oder** ***_os_ev_signal()*** ➔beide verwenden EventID ➔ggf. **anlinken nötig**

  - ►**Signalisiere Event** ***_os_ev_signal*** **(***event_id ev_id, int32 *value, u_int32 actv_flag***)**

Ablauf:

1. Das Event ***ev_id*** wird gesucht.
2. **neuer Zahlenwert N** des Events = **alter Zahlenwert A** + **Sinc.**
3. Die nächste in der Reihe wartende Task **W**, die auf den **Zahlenwert N** wartet wird aktiviert.
4. **Winc** wird zum Zahlenwert **N** des Events dazuaddiert wenn diese Task **W** gefunden und geweckt wurde:
   neuer Zahlenwert N = Zahlenwert N + Winc
5. Wenn ***actv_flag*** **== 1 und eine Task aktiviert wurde** gehe zu Schritt 3 sonst 6.
6. Der **Zahlenwert** N **des Events** wird in ***value*** gespeichert.

►Wahlweise werden ggf. nur die erste Task oder alle aktiviert!!
►Die Ausführung der Winc - Addition löst weitere Aktion aus wenn actv_flag = 1 !!

  - ►**Signalisiere und setze den Zahlenwert eines Event auf einen absoluten Wert**

_os_ev_set *(event_id ev_id, int32 *value, u_int32 actv_flag)*

Ablauf: wie *_os_ev_signal()* bis auf Schritt 2 ►Anstelle Sinc wird der **Zahlenwert** N des Events auf den übergebenen ***value*** Wert gesetzt.

- **Event freigeben :** ***_os_ev_unlink*** *(event_id ev_id);* *reduziert den* ***Linkcount um eins***

- **Event löschen:** ***_os_ev_delete*** *(char*event_name);* wenn Linkcount 0 ist ➔löschen des Events
- **Warten auf das Auftreten eines Eventereignisses *_os_ev_wait()/_os_ev_waitr()*** ►vorheriges **Anlinken des Events nötig!**

  Ablauf: ***_os_ev_wait (*** *event_id ev_id, int32* value, signal_code* si, int32 min_val, int32 max_val);*

  1. **Zahlenwert (Z) des Events *ev_id*** wird verglichen:
     a. **Falls *min_val <= Z <= max_val*** *erfüllt ist, wird* ***Winc addiert*** und ***Rückkehr aus _os_ev_wait()****.*
     b. Sonst: **Task wird in die Warteschlange (event queue) eingereiht**

  2. Späterer Zeitpunkt

     ►Event wird von anderer Task gesetzt und nun sei der gesetzte **Zahlenwert (Z) im Warte-Bereich**
     ►**Die erste Task die auf den Zahlenwert in der Queue wartete wird geweckt, in die aktive Queue eingereiht und Winc addiert**
     !!! **Ausnahme ►Die Task wird auch durch Empfang eines ´Signals´ geweckt**
     ►**Test der Ursache des Weckens nötig!!!**

  Bei Rückkehr wird in *value* der Eventwert gespeichert, der zum Weckzeitpunkt bestand:

  ►**wenn nun ein Signal die Weckursache war, muss eine Signalinterceptroutine existieren. Dann liegt der Eventwert bei Rückkehr aus *_os_ev_wait()* nicht im erwarteten Zahlenbereich (*min_val,max_val*).**
  ►**In *signal_code si*** wird im Falle Weckens durch einen Signalempfang die letzte empfangene Signalnummer hinterlegt.

- ***_os_ev_waitr()*** ►wartet auf das Erreichen eines Wertes relativ (Offset) zum gerade gültigen Wert des Events

OS9-Shellkommando events (Display Active System Events)

- **events** dient zur Anzeige der im System befindlichen events.

$ **events**

```
Event ID        Owner       Perm        Value       W-inc  S-inc    Links     Name
--------        -------     ----        ----------  ------ ------   ------    ------------
00010001        0.0         0003        0           0      0        1         sysmbuf
00030003        0.0         0003        0           -1     1        1         spf_rx
```

$ **events –h** ==> Anzeige Value in hex
$ **events –k**=name ==> Killen des Events name

Event ID	Event ID number
Name	Name of the event
Owner	Owner of the event
Perm	Event's permission field. For example, 0333 represents an event with all permissions set. Permissions are from left to right: reserved, public, group, owner.
Value	Current contents of the event variable
W-inc	Wait increment. Assigned when the event is created and does not change
S-inc	Signal increment. Assigned when the event is created and does not change
Links	Event use count. When the event is created, links is assigned the value one. links increments each time a process links to the event

Synchronisation mittels Signal und Sleep (hier: OS9000 Signale)

- **Signale** sind sog. Software-Interrupts die programmgesteuert von einer auslösenden Task an eine oder alle Anwendungstasks gesendet werden können.
- Ein **Signal besteht aus Signalnummer (signal-code)** und **EmpfängertaskID.**
- **OS9000-Signalnummern** sind systemweit gültig und sind wie folgt belegt:

1	**Wake-up signal**. Sleeping/waiting processes receiving this signal are awakened, but the signal is not intercepted by the intercept handler. Active processes ignore this signal. A program can receive a wake-up signal safely without an intercept handler. The wake-up signal is not queued.
2	**Keyboard abort signal**. When <control>E is typed, this signal is sent to the last process to perform I/O on the terminal. Usually, the intercept routine performs exit(2) when it receives a keyboard abort signal
3	**Keyboard interrupt signal**. When <control>C is typed, this signal is sent to the last process to perform I/O on the terminal. Usually, the intercept routine performs exit(3) when it receives a keyboard interrupt signal.
4	**Unconditional system abort signal**. The super user can send the signal to any process, but non-super users can send this signal only to processes with their group and user IDs. This signal terminates the receiving process, regardless of the state of its signal mask, and is not intercepted by the intercept handler.
5	**Hang-up signal**. SCF sends this signal when the modem connection is lost.
6-19	Reserved
20-25	Reserved
26-31	User-definable signals that are deadly to I/O operations.
32-127	Reserved
128-191	Reserved
192-255	Reserved
256-4294967295	User-definable non-deadly to I/O signals.

- Signal x kann mit dem OS9000-Kommando **kill <x> <EmpfID>** an Tasks verschickt werden.
- Signal 2/3 wird durch Eingabe ^e und ^c durch den SCF-Devicetreiber erzeugt **und** an die Task geschickt die **zuletzt** eine Ausgabe am Terminal getätigt hat!!
- Signal 1 hat eine Ausnahmefunktion! Signal 1 ist das WAKE-UP Signal und führt dazu dass die betreffende Empfängertask geweckt wird. **!!Achtung in diesem Fall wird die Signal-Empfangsroutine nicht durchlaufen!!**
- Ab Signal 256 kann jeder Benutzer eigene Signale belegen.
- **Task-Abort** mittels Signal 4 möglich, wird an den Empfänger !nicht! zugestellt und kann von einer User-State Task nur an Tasks der eigenen Group/Owner ID geschickt werden. Dann terminiert die Empfängertask. Super-User Tasks können an alle schicken. → interaktiv: **kill <EmpfID>**
- OS9: **Existiert keine Signal-Empfangsroutine und wird ein Signal geschickt, so terminieren user-mode Tasks**/system-mode Tasks nicht. **(Ausnahme: Gilt nicht für signal 1!)**
- **➔OS9000: Broadcast ein Signal an mehrere ►Destinationtask pid =0 ►Signal an alle mit gleicher user/groupID**
- Damit die **Empfängertask ein Signal** empfangen kann, stellt sie eine **Signal-Empfangsroutine** bereit und läßt diese **vom Betriebssystem als sog signal-intercept-Routine registrieren**.

- **Signal senden**►aktive Task verschickt ein Signal

  - ►das **RBS führt mit temporären Taskwechsel** zur Empfangstask **die** registrierte **Signalempfangsroutine des Empfängers sofort** aus. ►Die signal-intercept-Routine des Empfängers sollte demnach so kurz wie möglich sein.

  - ►die angesprungene Task wertet in der Interceptroutine den ihr übergebenen Signalcode aus und setzt z.B. eine eigene **globale-statische Variable (➔volatile deklariert, damit keine Compiler-Optimierungen wg. Registerverwendung!!)** damit nach Verlassen der Interceptroutine der Wert noch vorhanden ist. **►Rückkehr aus Interceptroutine besonders: _os_rte()!! ►Rückkehr zur Sendertask**

    - **Fall 1**: **Die bereite Empfangstask befindet sich in diesem Augenblick irgendwo im Taskcode und wartet nicht explizit auf das Auftreten eines Signals. (programmgesteuerte Synchronisation)**
      - ►Wenn die Task an die Reihe kommt (scheduled wird) kann programmgesteuert vom Task-Programmcode (außerhalb der signal-Interceptroutine) die globale Variable abgefragt und verarbeitet werden.
      - ►Programmgesteuerte Abfragemethode ►Ort und Zeit der Abfrage in der Hand des Programmierers
      - ► Nachteil: mehrfache Abfragen sind im Programmcode eingestreut ►ohne hinreichende Vorkehrung durch entsprechende Progammierung ggf. Verlust eines gleichen Signals durch Zeitabstand zwischen zwei Abfragen.

    - **Fall 2**: **Die Empfangstask hatte sich mittels _os_sleep()-Aufruf suspendiert und die Schlafdauer (timeout) ist noch nicht abgelaufen. *(Beachte: Der Aufruf von _os_sleep() setzt die Signalempfangsmaske auf NULL und erlaubt somit sofort den Empfang von Signalen)***
      - OS9: Sie ruht in der sog sleep-Queue, die nach der Restschlafdauer -kürzeste zuerst- sortiert ist, und wartet auf das Ablaufen ihres einstellbaren sleep-Timers. (Timerwert=0 ►warten unendlich)
      - ►OS9: Empfängt eine Task in der sleep-Queue ein Signal oder läuft ihr Timer ab so wird sie automatisch sofort in den Zustand bereit versetzt. Ihre Priorität und somit Rangstufe in der Bereit-Warteschlange ergibt sich bei Nicht-Echtzeittasks aus dem age.

- ►OS9: Die Task wird nach _os_sleep()-timer Aufruf fortgesetzt ►wenn der zurückgegebene Zeitwert ungleich 0 ist, so wurde durch ein Signalempfang geweckt. ►Empfangener Signalcode wird mit zurückgegeben.

- ►OS9: Um kritische Abschnitte im Taskcode gegen das Stören durch Signalunterbrechungen zu schützen gibt es vergleichbar der Interruptmaske eine Signalempfangsmaske. Ist sie Null so ist der Signalempfang möglich.
  ➔ ***_os_sigmask(1)*** erhöht die Signalempfangsmaske um Eins. Ist danach die Signalmaske grösser eins, so ist der Signalempfang gesperrt;
  ➔ ***_os_sigmask(-1)*** erniedrigt die Signalempfangsmaske um eins, wenn diese nun Null ist, ist der Signalempfang wieder möglich
  - die Aufrufe ***_os_sigmask(0), _os_wait()*** oder ein Aufruf von ***_os_sleep()*** setzen die Signalempfangsmaske auf den Wert Null und **entsperren für den uneingeschränkten Signalempfang** .

- **➔ Ist der Signalempfang in einer Task durch den Wert Null in der Signalempfangsmaske und das Vorhandensein einer registrierten Signalintercept möglich, so führt OS9 folgende Schritte aus:**

  - OS9: Vorbereiten des Stacks und der Aufrufumgebung der zu rufenden Taskumgebung
  - ***OS9: _os_sigmask(1)***
  - OS9: Aufruf der Taskinterceptroutine der Task, erster und einziger Parameter die Signalnummer
  - UserTask: Die Taskinteceptroutine beendet sich mit *_os_rte()*
  - **OS9: In der Endebehandlung durch OS9 Aufruf *_os_sigmask(-1)***
  - OS9: Aufräumen des Stacks und Rückkehr zu voherigen Zustands

- **Beachte: Wird in der Signalinterceptroutine *_os_sleep() oder sigmask(0)* aufgerufen, so wird die Signalempfangsmask auf Null gesetzt und im Signalempfangsvorgang der rekursive Aufruf der Signalinterceptroutine erlaubt und ist auch seitens OS9 *erlaubt.* Die Rekursivität im Aufruf ist in diesem Fall durch die eigne Implementation in der Signalinterceptroutine zu berücksichtigen**

- **Hinweis:** Im oben genannten Fall 2 ist man nach Rückkehr aus dem durch den Signalempfang geweckten *_os_sleep()* bereit für den nächsten Signalempfang, da *_os_sleep()* die Signalmaske auf NULL setzt. Soll bei der Rückkehr aus *_os_sleep()* aber der Signalempfang gesperrt sein, so kann man in der eigenen Signalintercept-Routine mittels des Aufrufs *_os_sigmask(1)* die Signalempfangsmask um eine weitere Eins erhöhen, so dass durch das *_os_rte() und der Ausführung von sigmask(-1) durch OS9* die Signalmaske grösser Null ist und damit weiterhin gegen weiteren Signalempfang gesperrt bleibt.

- **►Signale werden sequentiell an die Empfänger weitergereicht. ➔für jedes Signal wird je einmal die Signal-Interceptroutine aufgerufen.**

- **►Ob gesperrt oder nicht, unbearbeitete Signale werden in der Reihenfolge ihres Auftretens gequeued.**

- ***Beispiel Empfangstask:***

  #include <stdio.h> ...#include <signal.h>...#include <process.h>...#include <types.h>..#include <cglob.h>...#include <stdlib.h>..include <const.h>

```
volatile signal_code Empfangs_signal;
void  signal_empfang (signal_code signal)
{
Empfangs_signal = signal;
_os_rte();
}
main (int argc, char* argv[ ])
{
u_int32 zero; error_code myerr; signal_code si ;
if ( (myerr =_os_intercept ( signal_empfang, _glob_data) ) !=SUCCESS) exit (myerr);
                    /* Registrierung der Intercept_Routine*/
zero = 0 /* unendlich lang schlafen */ ;
if ( (myerr =_os_sleep(&zero,&si) ) !=SUCCESS) exit (myerr);
          /* hier: warten zero=0 unendlich auf ein Signal !!Standardrückkehrwert:
             ►zero ist verändert und enthält Restzeit bis Ablauf Timer */
printf ("Signal empfangen: %d oder %d \n",Empfangs_signal,si); /* welches war´s denn? */
}
```

- ***Beispiel Sendetask: (includes wie oben)***

```
if ( (myerr =_os_send ( (process_id) EmpfangsPID, (signal_code) Signalnummer)) !=SUCCESS) exit (myerr);
```

- **Timer-Alarme sind eine Sonderform der Nutzung des Signalmechanismus in Verbindung mit einer Uhr. Anstelle einer Signalsendetask schickt das RBS nach einer einstellbaren Zeit ein vorher vereinbartes Signal an eine Task (auch periodisch).**
  - OS9: Ein Alarm ist an eine Uhr gebunden. Nach Ablauf einer einstellbaren Uhrzeit erhält der Alarmauftraggeber ein frei vorgebbaren Signalcode gesandt. Alarmuhrzeiten können Absolutzeiten, Deltazeiten und auch relative Zeiten sein. Es sind auch zyklische Alarme möglich►Watchdog ähnlich.

    Alle Alarme benötigen: #include <alarm.h>

  - Zyklischer Alarm: ***error_code _os_alarm_cycle (alarm_id *alrm_id, signal_code signal,u_int32 time);***
    signal *wird nach* ***time*** *(i.d.R. in TICs)* ***periodisch*** *an die eigene Task geschickt*
    alarm_id *dient zur Identifikation dieses Alarms.*

  - Einmal Alarm: ***error_code _os_alarm_set (alarm_id *alrm_id, signal_code signal,u_int32 time);***
    signal *wird nach* ***time*** *(i.d.R. in TICs)* ***einmal*** *an die eigene Task geschickt*
    alarm_id *dient zur Identifikation dieses Alarms.*

  - Alarm Löschen: ***error_code _os_alarm_delete (alarm_id alrm_id);***
    Der Alarm ***alarm_id*** *wird gelöscht, damit wird kein Alarmidentifiziert durch* ***alarm_id*** *mehr geschickt.*

Tasksynchronisation durch Semaphore

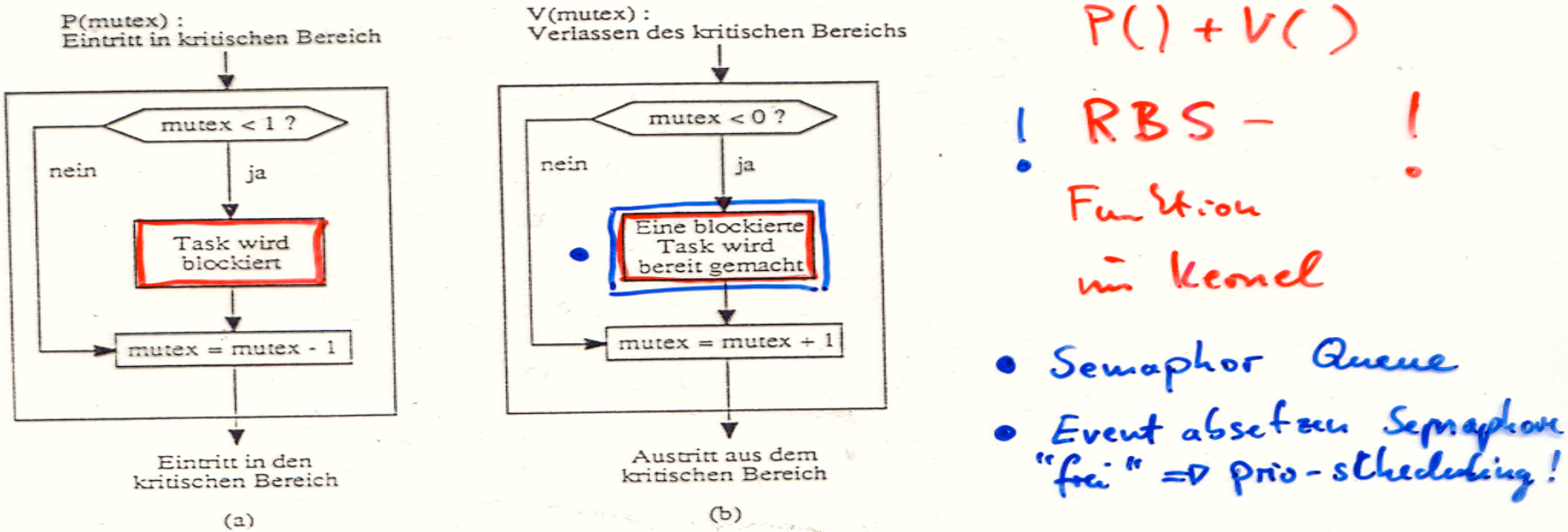


Abbildung 8.11: Die Anwendung der P-Operation (a) und der V-Operation (b) auf ein allgemeines Semaphor *mutex*. Diese Operationen sind nicht unterbrechbar was durch die Einrahmung angedeutet ist [Koch80]

► Theorie: Semaphore dient zur Steuerung des Zugriffs auf ein begrenzt verfügbares Betriebsmittel
einfachster Fall: Betriebsmittel nur einmal vorhanden ►nur eine Task (und ein Prozessor!) darf es gleichzeitig benutzen!! Struktur, mutex = mutual exclusion, P() und V() sind unteilbare Funktionen und müssen durch RBS realisiert werden (➔Multiprozessorsysteme sync mit anderen Prozessoren!!) ►Vertagung der wartenden Tasks durch RBS ►Warteschlange in der zeitlichen Ankunftsreihenfolge

Realisieren von Semaphoren mittels OS9-Events

- **Semaphor-Operationen P() und V()** müssen unteilbar in Einem ausgeführt werden. ➔ Signalisieren von und Warten auf Events ist eine unteilbare Operatiom. Die **Anzahl der exclusiven Betriebsmittel,** die es mittels eines Semaphors zu verwalten gilt, sei **n.**
- **Lösungsidee:**
  1. Anlegen eines Events **z.B.: "Sema1"**
  2. Vorbelegen des **Event-Wertes** mit **n**
  3. **Winc** auf **-1**
  4. **Sinc** auf **1**

➔ **Belegen P() -passieren-:**Der Benutzer will ein Betriebsmittel belegen und führt als P() einen ***_os_ev_wait()-***Aufruf mit den Parametern Minval=**1** und Maxval=**n** durch.
➔ Ist der Event-Wert im angegebenen Bereich -also eins der n Betriebsmittel frei- so kommt der Aufruf sofort zurück. Es wird aber **Winc=-1** auf den Eventwert angewendet ➔ somit ist ein Betriebsmittel weniger frei.
➔ Ist der Eventwert nicht im Bereich, also 0, so wird der Aufrufer suspendiert.

➔ **Freigeben V() -verlassen-:** Der Benutzer gibt ein Betriebsmittel frei und führt als V() einen ***_os_ev_signal()-***Aufruf durch. (*actv_flag = 0/1 ➔ egal weil winc nach dem erfolgten Wecken des ersten Wartenden angewendet wird und damit ggf. wieder 0 ist, und kein weiterer geweckt wird)*
➔ **Sinc=1** wird einmal auf den Eventwert addiert
➔ War der Eventwert vorher 0 und es warten welche auf ein Betriebsmittel, so wird der Nächste geweckt

- **!Achtung P() und V() müssen sich pärchenweise in jeder Applikation abwechseln!!**
- **Sollte der Eventwert kleiner 0 oder größer n werden, so liegt ein Fehler vor ➔ z.B. Überwachungstask bewacht den ungültigen Bereich und meldet Fehlbenutzung!!.**

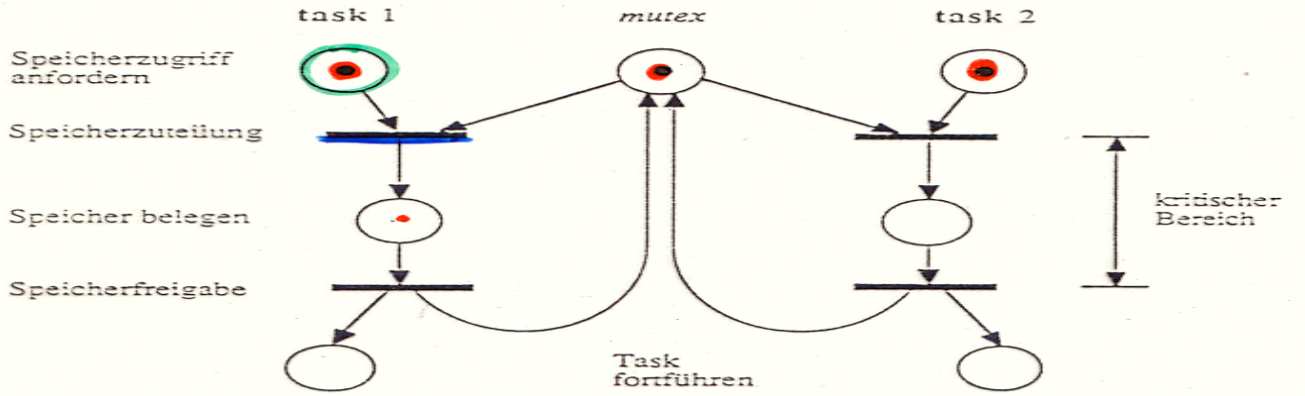


Abbildung 8.12: Petri-Netz-Darstellung des wechselseitigen Ausschlusses mittels des Semaphors *mutex*

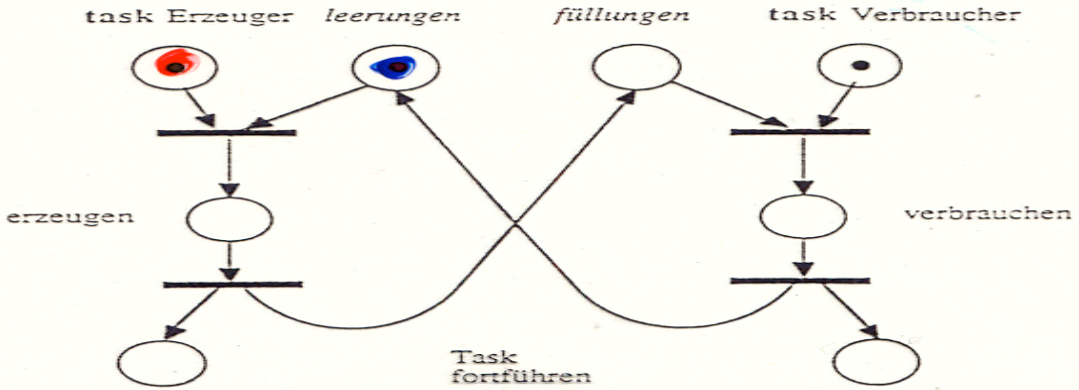


Abbildung 8.13: Darstellung des einfachen Erzeuger/Verbraucher-Problems mit Hilfe eines Petri-Netzes

Bild : Synchronisation einer Erzeuger und einer Verbraucher Task!! ►zwei Semaphore nötig: Leerung und Füllung, wiederum einfachster Fall: ein Platz zum Befüllen►Erzeuger muss warten bis Verbraucher abgeholt hat ►Praxisprobleme: durch Designfehler der SW ►Verklemmungen!! beide warten aufeinander!! ►Sonderfall: sog Monitore sind gemeinsam nutzbarer Code und Daten ►sie werden den Tasks mittels Semaphore exclusiv zugeteilt. Will gleichzeitig ein anderer in den Monitor eintreten so muss er warten auf seine Freigabe

Petri-Netze

- **Petrinetze sind ein Hilfsmittel zu Darstellung nebenläufiger Prozesse mittels gerichteter Graphen**
- **Petrinetze bieten die Möglichkeit übersichtlich Synchronisationsaufgaben zwischen beteiligten Prozessen zu veranschaulichen und festzulegen. Es gibt:**

- **MARKEN, STELLEN und TRANSITIONEN**

  - **STELLEN und TRANSITIONEN haben Ein- und Ausgänge**
  - **STELLEN und TRANSITIONEN werden über unidirektionale gerichtete Pfeile miteinender verbunden**
  - **STELLEN und TRANSITIONEN wechseln sich ab; es werden NIE zwei TRANSITIONEN und/oder STELLEN aufeinenderfolgend verbunden.**

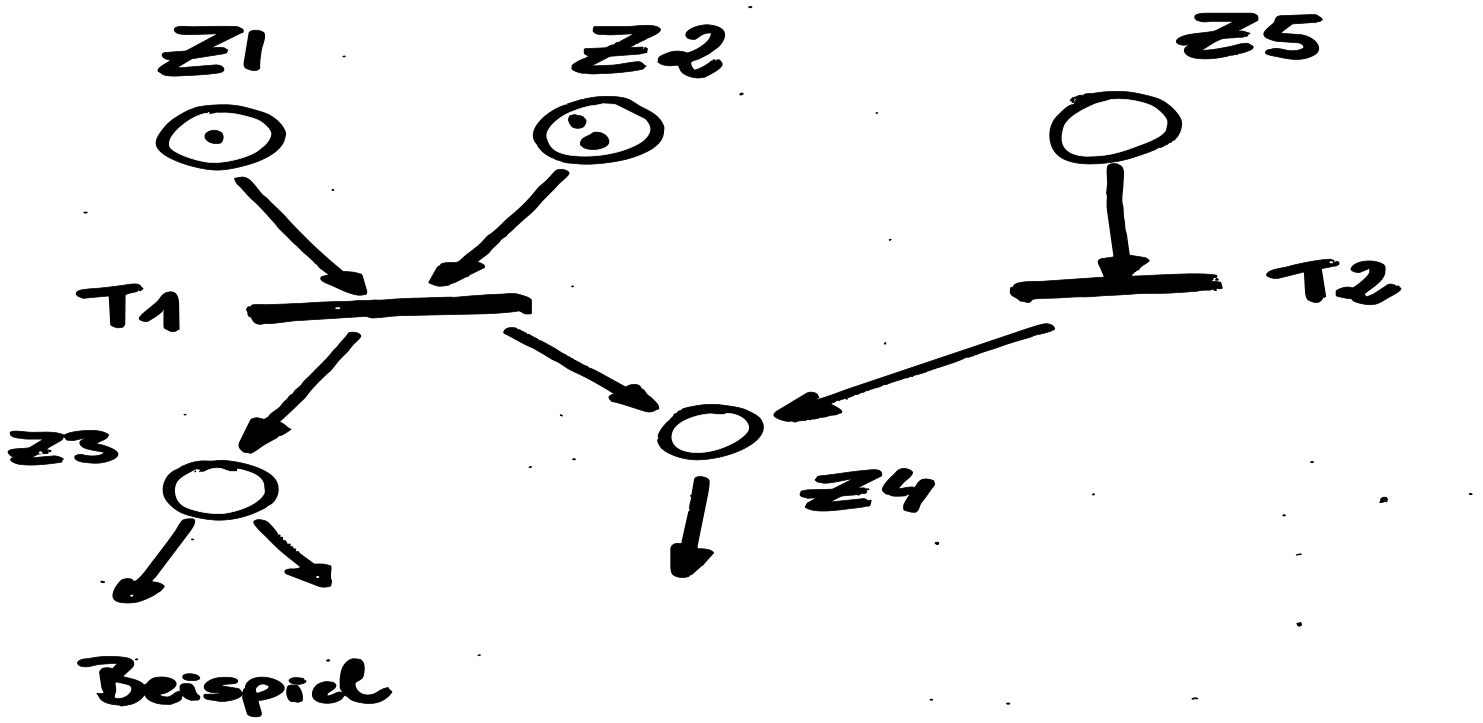
- **Die STELLEN entsprechen den Zuständen, die ein Prozeß durchlaufen kann**
  - **aktive STELLEN enthalten MARKEN, inaktive keine**
  - **jede STELLE besitzt eine MAXIMALE Aufnahmekapazität von MARKEN, default: unendlich viele Marken möglich**

- **TRANSITIONEN stellen (Weiterschalt-)Bedingungen, um den Prozeßzustand von STELLE zu STELLE weiterzuschalten.**
  - **➔ Diese Bedingungen können als boolsche Gleichungen formuliert sein.**
  - **➔ Die Bedingung verknüpft alle über die gerichteten Pfeile angeschlossenen STELLEN**
  - **➔ TRANSITION nur aktiv wenn Bedingungen erfüllt**

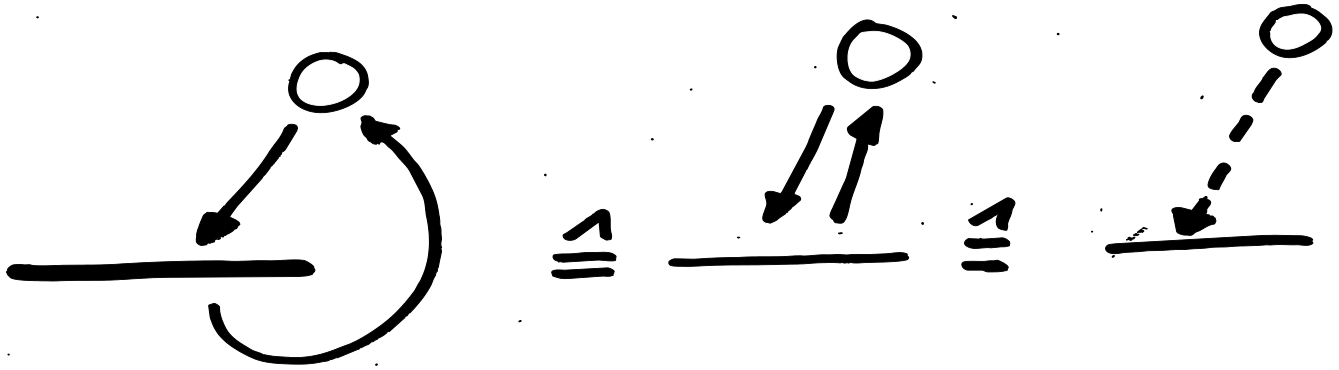
- **SCHALTREGEL:**

➔Jede TRANSITION schaltet automatisch, wenn jede einzelnen STELLE, die mit dem Eingang der TRANSITION verbunden ist, mindestens EINE MARKE enthält UND die TRANSITIONS-Bedingung erfüllt ist.
➔ **Im Schaltvorgang wird jeder einzelnen STELLE, die am Eingang angeschlossen ist, EINE MARKE abgenommen**
➔ **Jede am Ausgang der Transition angeschlossenen STELLE erhält genau EINE MARKE (es gibt unendlich MARKEN!!)**
- ➔ **STELLEN die vorher keine MARKE hatten werden dann aktiv!**
- ➔**STELLEN die dabei alle MARKEN verlieren werden inaktiv!**
- ➔ **(theoretisch) alle Transitionen schalten ‚gleichzeitig nebenläufig'**

- **Beispiel:**



- **Abkürzungen:**



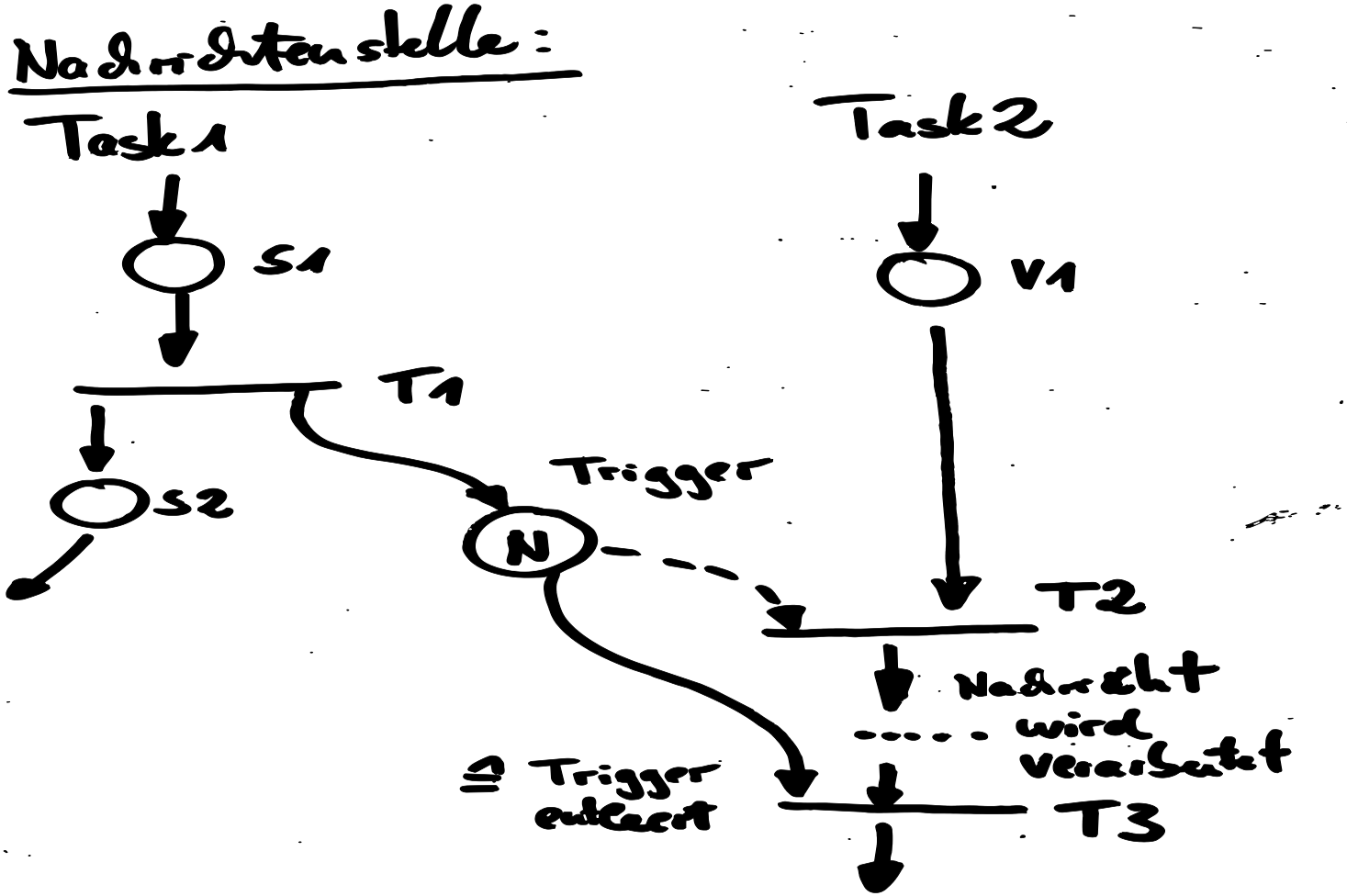
- **Petrinetze lassen sich leicht formal auf Konsistenz überprüfen**

  ➔ Prüfung ob alle STELLEN je MARKEN erhalten (also aktiv sind)

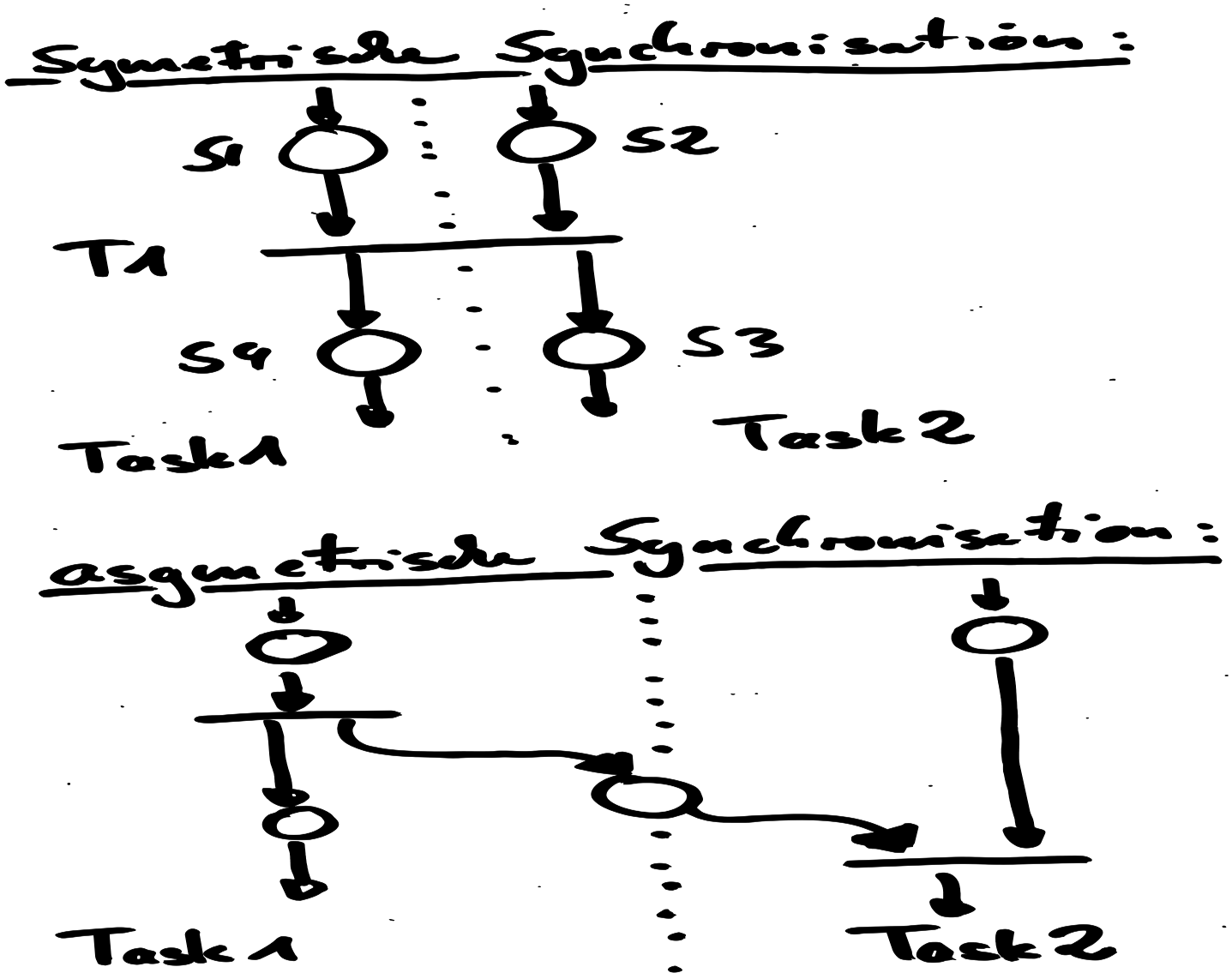
  ➔Verklemmungsdetektion: gehen Marken verloren, oder finden keine TRANSITIONEN statt

  ➔Kapazitätsüberschreitung: Marken werden produziert (➔ siehe Ausgang TRANSITIONEN) und nicht verbraucht; es wird die STELLEN-Kapazität überschritten

- **Beispiel 2**


Nachrichtenstelle:
Task 1
S1
T1
S2
Trigger
N
Task 2
V1
T2
Nachricht
wird
verarbeitet
Trigger
auslöst
T3

Beispiel 3:


Symetrische Synchronisation:
S1
S2
T1
S4
S3
Task1
Task2
asymetrische Synchronisation:
Task1
Task2

Synchronisation zwischen Threads

- Der Zugriff auf eine gemeinsame Variable von zwei Threads aus muss um Dateninkonsistenzen zu vermeiden mittels des sog. Semaphor-Konzepts serialisiert werden. D.h. zu einer Zeit darf nur ein Thread Zugriff auf die gemeinsame Variable besitzen:
  - Vor dem Zugriff auf eine gemeinsame Resource (z.B. Variable) muss die Semaphore von **jedem** der zugreifen will belegt werden.
  - **Zwischen** dem **Belegeaufruf** der Semaphore **und deren Freigabeaufruf kann** auf die zu schützende Resource **exklusiv zugegriffen werden**, da garantiert ist dass alle weiteren Zugriffsversuche beim Versuch des Belegens der Semaphore vom System geblockt werden.
- Semaphore, **taskübergreifend**, können mittels **OS9000-Events** oder über **OS9000-Semaphorimplementation** realisiert werden.
- Semaphore **innerhalb** einer Task zwischen allen innerhalb dieser Task befindlichen Threads können resourcensparend über den **pthread-Mutex (**Mutex) Mechanismus realisiert werden.
- Ein **Mutex** ist eine Variable,
  - die in der Task einmal angelegt werden und
  - von allen nutzenden Threads aus **sichtbar** sein muss,
  - die nach der **Initalisierung** durch Funktionsaufrufe von Threads **belegt** und wieder **freigegeben** wird.

  - Wird von einem anderen Thread versucht einen bereits **belegten Mutex** zu **belegen**, dann wird dieser **automatisch vom System solange suspendiert** bis der Mutex wieder freigegeben wird. Der nächste auf Belegung wartende Thread wird geweckt und **gleichzeitig in einem atomaren Schritt** wird der Mutex von diesem Thread belegt.
- **Anlegen eines Mutex:**
  - Definition einer Mutexvariable mit Defaultinitialisierung (frei):
    pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
  - Alternative Initialisierung durch Aufruf **pthread_mutex_init()** möglich.

- **Belegen eines Mutex:**
  - Mittels **int pthread_mutex_lock(pthread_mutex_t *mutex);** wird der Mutex **mutex** belegt, wenn er noch frei war ansonsten wird der Aufrufer solange suspendiert bis der Mutex wieder frei ist und alle vor ihm in der Warteschlange bedient wurden.

- **Freigeben eines Mutex:**
  - Mittels **int pthread_mutex_unlock(pthread_mutex_t *mutex);** kann der Mutex **mutex** von dem Thread, der vorher belegt hatte, wieder freigegeben werden.

- **Testen und ggf. Belegen eines Mutex in einem Schritt:**
  - Mittels **int pthread_mutex_trylock(pthread_mutex_t *mutex);** wird der Mutex **mutex** getestet ob er belegt ist, wenn er noch frei war wird er belegt, ansonsten wird zum Aufrufer mit entsprechender Fehlermeldung EBUSY zurückgekehrt; d.h. er wird nicht blockiert.

- **Anmerkung:** Es stehen zur Threadsychronisation auch sog. pthread-Condition-Variable und deren Steuerung für Threads innerhalb einer Task zur Verfügung. Die Implementation ist nur eingeschränkt funktional und sollte nur bei Portierung von Programmen nach OS9000 Verwendung finden. Der OS9000-Eventmechanismus ist semantisch gleichwertig, einfacher zu benutzen und funktioniert auch intertaskweit.

Threadglobale und threadprivate Variable

- Alle Threads einer Task sehen alle C- **globalen** Variablen. Diese werden zur Programmstartzeit einmal initalisiert und existieren während der Tasklaufzeit.

- Alle Threads einer Task, die Funktionen in einer Übersetzungseinheit (ein C-Source-File) nutzen, sehen alle C- **globalen statischen** Variablen die in dieser Übersetzungseinheit angelegt wurden. Diese werden zur Programmstartzeit einmal initalisiert und existieren während der Tasklaufzeit.

- Alle **funktionslokalen** Variablendefinitionen innerhalb einer Funktion sind **threadprivat**, da sie auf dem Stack bei Funktionsaufruf dynamisch angelegt werden. Jeder Thread hat seinen eigenen Stack. Beim Verlassen der Funktion werden diese lokalen Variablen 'vernichtet'.

- Alle **funktionsstatischen** Variablen sind in der Funktion in der sie definiert sind **einmal** angelegt und existieren bis zum Taskende. D.h. diese Variablen existieren auch nach Verlassen der Funktion weiter. **Fatal:** Wird ein Verweis auf eine solche Variable nach ausserhalb der Funktion in der sie definiert wurde exportiert, so gilt für die Datenkonsistenz dasselbe wie für globale Variable.

- **Fazit: Alle Zugriffe von Threads innerhalb einer Task auf gemeinsam genutzte Variable, die nicht funktionslokal sind, müssen synchronisiert werden (z.B. durch Semaphore) und Dateninkonsistenzen zu vermeiden.**

- **Wie realisiert man globale aber threadprivate Variable?**
  - D.h. **Variable die ausserhalb von Funktion** für den C-Programmcode sichtbar aber threadprivat -wie funktionslokale Variable- sind.
  - Die Sprache C kennt das Konzept der Threads nicht. Also gibt es keine C-Speicherklasse die diese Anforderung erfüllt.

- Es kann ein Thread in einer C-Funktion **programmgesteuert** eine **threadprivate Variable** anfordern, die vom System verwaltet wird.
  - Jede threadprivate globale Variable wird zur Laufzeit programmgesteuert einmal erzeugt und über einen task- und threadweit eineindeutigen Keywert identifiziert. Alle threadprivate globale Variable haben demzufolge unterschiedliche Keywerte.
  - Eine threadprivate globale Variable ist vom Typ generischer Zeiger
  - Den Zugriff auf eine threadprivate globale Variable threadvar kann man sich wie folgt als Pseudo C-Code vorstellen:

```
struct Threadvararray {
                        void * threadvar [MAXTHREADIDNUMMER]
            };
struct Threadvararray * Keywert;
```

Erzeugen der **threadprivater Variable**: **Keywert =**
(struct Threadvararray *) new (struct Threadvararray);
Zugriff in einem Thread mit ThreadID lesend:
threadvar = Keywert->threadvar[ThreadID];
Zugriff in einem Thread mit ThreadID schreibend:
Keywert->threadvar[ThreadID]= threadvar;

►Ablaufskizze des Anlegens eines threadprivaten Speicherbereichs mit Anwendung:

►Anlegen einer sog. **threadspezifischen Keyvariable,** die eine threadprivate Variable 'aufnehmen' kann. Definition der 'leeren' Variablen: **static pthread_key_t key;**

►Mittels **int pthread_key_create(pthread_key_t *key, void (*destructor) (void *));**

- wird ein **einzigartiger** Keywert der die threadprivate Variable darstellt erzeugt,
- und unter **key** abgelegt und
- Speicherplatz vom System unter diesem Keywert reserviert, der die threadprivaten Zeigerwerte hält.
- Wird der Key wieder freigegeben, so wird die Funktion auf die **destructor** zeigt aufgerufen.

Dieser Aufruf erfolgt genau einmal pro Erzeugung einer threadprivaten Variable

► Mittels **int pthread_key_delete(pthread_key_t key);** wird der Keywert gelöscht und der vereinbarte Destructor beim Erzeugen aufgerufen.

► Seinen threadprivaten Wert **value** kann der aufrufende Thread **belegen** mittels:
int pthread_setspecific(pthread_key_t key, const void *value);
Die Aufrufer ThreadID und der Keywert in **key** sind der Schlüssel für die Ablage des Wertes.

►Seinen threadprivaten Wert als Funktionsrückgabewert kann der aufrufende Thread **lesen** mittels: **void *pthread_getspecific(pthread_key_t key);**
Die Aufrufer ThreadID und der Keywert in **key** sind der Schlüssel für die Abfrage des Wertes.

►Der eigentlich benötigte threadprivate Speicherplatz muss über **malloc()**-Aufrufe durch die Threads selbst auf dem Heap erzeugt werden. Der vom **malloc()-**Aufruf zurückgegebene Zeigerwert wird dann mittels des **pthread_setspecific()-**Aufrufs threadprivat aufbewahrt.
►Der erste Aufruf der Funktion **int pthread_once()**in einer Task führt eine angebare Aktion genau einmal durch. Jeder weitere Aufruf dieser Funktion führt nichts aus. Damit lassen sich Keywerte sicher einmal und nur einmal erzeugen.

E 5.4 Intertaskkommunikation

►In RBS müssen Prozesse programmgesteuert untereinander Nachrichten austauschen können.

►Das RBS stellt i.d.R. hierfür Mechanismen und damit verbundene Dienste, die über die SYSA SS erreichbar sind, bereit.

Messagebuffers

(Listen aus Poolelementen) (nicht OS9000)

►Nachrichten (messages) werden durch die Sender-Task in Poolelementen (buffers) abgelegt und an eine Empfänger-Task verschickt durch Abgabe der Nachricht an den Sendedienst des RBS.
►Empfangsadresse ist die Empfangs TaskID. ►RBS kopiert die Nachricht in einen internen Puffer und trägt sie in eine Verwaltungsliste ein. ►Das RBS bewahrt die messages bis zur Abholung durch den Empfänger auf.
►Messages in Mehrprozessorsystemen und vernetzten Systemen werden verschickt und beinhalten neben TaskId auch Empfangsrechnerkennung => RechnerID + TaskID hier dann Symbolisch!! RBS kann dann physikalische Adressen ermitteln und auch den Kommunikationspfad
► **oft:** Empfänger pollt nach Nachrichten
► Die Verwaltung im RBS geschieht meist in einer über Zeiger verkettete Liste ► mehrere messages für eine Task möglich
► Liste: wird aus der Sicht des Empfängers nach FIFO (= Beibehaltung der Sende-Zeit-Reihenfolge) oder LIFO (Prinzip =Neueste Nachricht zuerst) geführt.

Globaler gemeinsamer Speicher (memory mailbox via shared memory)

►Gemeinsamer Speicherbereich (shared memeory) vom RBS angelegt und allen Partnern zugänglich gemacht entweder:
- ►im RBS Datenbereich (selten)
- ►gemeinsamer Daten-Speicherbereich (OS9000:Datenmodul)

►Zugriffssteuerung der einzelnen Tasks auf gemeinsamen Speicherbereich über EVENTS oder Semaphore!!

►RBS blendet das Datenmodul an unterschiedlichen logischen Adressen bei den beteiligten Tasks ein.
- ►MMU setzt logische in gemeinsame, gleiche physikalische Adressen um
- ►Umsetzung so, dass Datenmodul von allen beteiligten Tasks, die Zugriff haben, sichtbar.

► Multiprozessorsysteme mit Netzkoppelung ohne gemeinsamen Arbeitsspeicher,
- ►hat jeder z.B. eine Kopie des Datenmoduls. ►Schreibt einer auf seine lokale Kopie, so erhalten die anderen ein Update ►Problem der Synchronisation bei Netz- oder Einzelrechnerausfall

Shared-Memory (OS9-Datenmodul)

- **OS9 unterstützt im Entwicklerkernel Speicherschutzmechanismen mittels MemoryManagementUnit(MMU)** ►Realisierung von sog. **shared-memory** erfolgt hier mittels sog. **Datenmodulen.**

- **Datenmodul** ist eine Sonderform des allgemeinen Moduls ►Modul-Type **MT_DATA ;** Language **ML_ANY**

- Das **Datenmodul** kann als Datei vorhanden sein oder es wird (meistens) von einem Prozess generiert.

- Ein **Datenmodul** ist gekennzeichnet durch einen **Modulnamen** und seine **Größe.** Die innere Strukturierung der Daten bleibt dem Nutzer überlassen. ►Nach dem Anlegen des Moduls wird der Datenbereich gelöscht! ➔Der Erzeuger erhält **Zeiger** auf den **Modulkopf** und den **Moduldaten.** ►***_os_datmod()***

- Ein **Datenmodul** besitzt ebenfalls ein CRC Abschluß. ►Soll das **Datenmodul** in eine Datei gespeichert werden **und** wieder ladbar sein ►CRC Berechnung durchführen **bevor** Abgespeichert wird ►***_os_setcrc()***

- Neben dem Erzeuger des Datenmoduls kann sich jeder, der entsprechende Zugriffsrechte (➔Modulpermissions) besitzt, Zugriff durch **Anlinken** verschaffen (►Modulname dient zur Identifikation) . ►Jeder Link erhöht den **Datamodullinkcount** ►Jeder angelinkte erhält einen **Zeiger** auf den Modulkopf/Moduldaten. ➔***_os_link()***

- Die **synchronisierte Zugriffssteuerung** auf das ggf. **gemeinsame Datenmodul** bleibt den **Tasks** überlassen. ►OS9-**Eventsteuerung, allg. Semaphore**

- Der Erzeuger und alle angelinkten Benutzter müssen das Datenmodul wieder durch unlinken freigeben. ►***_os_unlink()***

- ***Beispiel Erzeugen eines Datenmoduls:***

```
#include <stdio.h>  #include <module.h> #include <types.h> #include <errno.h> #include <const.h>
#include <memory.h>

#define DATA_SIZE  1024                    /*► Größe des Datenmodulbereichs!!*/
#define DMOD_NAME "MeinDatenModul"  /*► Datenmodulname */
void main (int argc, char* argv[])

int * meine_daten                          /*► später: Zeiger auf den Beginn des Datenbereichs*/
u_int16 attr_rev, type_lang;
u_int32 perm;
error_code err
mh_data *mod_head;

type_lang = mktypelang (MT_DATA,ML_ANY);
attr_rev = (MA_REENT <<8);

perm = MP_OWNER_READ | MP_OWNER_WRITE | MP_GROUP_READ | MP_GROUP_WRITE
|MP_WORLD_READ|MP_WORLD_WRITE ;

err = _os_datmod ( DMOD_NAME, DATA_SIZE * sizeof (int), &attr_rev,
                   &type_lang, perm, (void**) &meine_daten, &mod_head,MEM_ANY);
```

► Zeiger: meine_daten zeigt jetzt auf den Beginn des Datenbereichs. mod_head zeigt auf den Modulkopf

- ***Anlinken auf ein bereits erzeugtes Datenmodul:***

```
►includes wie vorher!

#define DMOD_NAME "MeinDatenModul"  /*►Datenmodulname */

void main (int argc, char* argv[])

char* d_name= DMOD_NAME;
int * meine_daten = NULL;        /*►später: Zeiger auf den Beginn des Datenbereichs*/
u_int16 attr_rev, type_lang;
error_code err
mh_com *mod_head;

type_lang = mktypelang (MT_DATA,ML_ANY);

err = _os_link ( &d_name, &mod_head, (void**) &meine_daten, &type_lang, &attr_rev ) ;
```

►*meine_daten zeigen auf den Beginn des Datenmoduls* ►*Die Datenmodul-Größe muss bekannt sein!*

- ***Freigeben eines angelinkten Datenmodul:***

```
►includes wie vorher
mh_com * mod_head; /*►zeigt auf den Modulkopf!! Vorher durch link/create bekommen!!
err =_os_unlink ( mod_head );
```

- ***Laden eines abgespeicherten Datenmoduls:***

```
#include <module.h>   #include <modes.h> #include <memory.h>  #include <types.h>
#include <const.h>

char * mod_name = DMOD_NAME;
mh_com * mod_head;                    /* Hier wird nach dem Laden der Verweis auf den Modulkopf hinterlegt*/
int* pdata ;                          /* zeigt nach laden auf das erste Datenbyte des Moduls*/
u_int32 mode; u_int16 type_lang ; u_int16 attr_rev ;

type_lang = mktypelang (MT_DATA,ML_ANY);
mode = S_IREAD| S_IWRITE| S_IGREAD|S_IGWRITE|S_IOREAD|S_IOWRITE ;
error_code  _os_load( mod_name,& mod_head,(void **)pdata , mode,&type_lang,&attr_rev, MEM_ANY);
```

- ***CRC-Ausrechnen***

```
mh_com * mod_head; /*► zeigt auf den Modulkopf!! Vorher durch link/create bekommen!!
_os_setcrc ( mod_head) ;
```

- ***Speichern eines Datenmoduls in eine Datei:***

```
Umweg: system ("save -f=<filename>  <modulname>");
```

Pipes

- **Pipes** dienen primär zur **unidirektionalen Daten-Kommunikation** zwischen **zwei oder mehr Prozessen**.

- **Ein oder mehrere Prozesse** sind **Sender** und **ein Prozess** der **Empfänger.** Das RBS **OS9** übernimmt in einer **FIFO-Queue,** genannt **Pipe,** die Zwischenspeicherung der Daten.

- Es gibt **unnamed Pipes** und **named Pipes.**

- **Unnamed Pipes** werden immer dann von OS9 angelegt wenn mehrere Kommandos mittels **!** verkettet werden. ►die Ausgabe des ersten Kommandos gelangt in ein unnamed Pipe von der das nach dem ! folgende Kommando die Daten als Eingabe einliest.

- **Named Pipes** tragen wie eine Datei einen **Namen** und sind bei OS9 auf dem **Device /pipe** abgelegt. Dieses Device läßt sich mittels der normalen Dateikommandos (dir -e /pipe ,list...) ansprechen. Auf dem Device /pipe existiert **kein** hierarchisches Directorysystem. Alle named-Pipes liegen direkt im sog. Root-Directory. Anstelle der Größe der Pipes wird beim *dir -e /pipes*-Kommando die Anzahl der im FIFO befindlichen Bytes angegeben.

- **Named//Unnamed Pipes in OS9 sind voreingestellt maximal 128 Byte groß ►programmgesteuert kann aber beim Anlegen von named-Pipes die maximale Größe frei gewählt werden.**

- ***Beispiel: Anlegen,Beschreiben und Anzeigen einer named-Pipe mittels mshell-Kommandos***

```
$dir -e /pipe

            Directory of /pipe  09:49:29
Owner   Last modified      Attributes    Sector     Bytecount     Name
    --------  ------------------      -----------    --------    ------------     -------

$echo   "Testdaten fuer die Named-Pipe hugo!!"  >/pipe/hugo

$dir -e  /pipe

            Directory of /pipe  09:50:10

Owner   Last modified      Attributes            Sector     Bytecount     Name
    --------  ------------------      ---------------       --------    ------------     -------
   0.0     03/10/01 09:50      --------------wr       372780             37      hugo

$list  /pipe/hugo

Testdaten fuer die Named-Pipe hugo!!

$dir -e /pipe
            Directory of /pipe  09:51:00
Owner   Last modified      Attributes    Sector     Bytecount     Name
    --------  ------------------      -----------    --------    ------------     -------
$
```

- **Pipes** sind Dateien die eine variable Länge haben►Der **Sender und der Empfänger** behandeln **Pipes** mit denselben Betriebssystemaufrufen wie normale Dateien. ►Eine **Pipe** muss, wie jede Datei, erzeugt/geöffnet/geschlossen werden ►**Filename: "/pipe/<mein_pipe_name>"** ►Es werden die üblichen Lese/Schreibaufrufe für Dateien verwendet ►Es gibt keinen Fileposition-Zeiger ►Der **Sender** schreibt beliebig in die Queue; die Daten werden an die Queue angehängt. ►Der **Empfänger** liest aus der Queue aus, wobei das älteste Datum zuerst gelesen wird.

- **Pipes, die leer sind und die von keinem Prozess geöffnet (benutzt) sind, werden von OS9 gelöscht!!!**

- **Named-Pipes unterliegen der Zugriffsrechtsvergabe wie alle Dateien.**

- **Besonderheiten beim Schreiben bzw. Lesen von named-Pipes:**

  1. **Schreiben auf eine volle Pipe ►Sender wird suspendiert bis wieder Platz ist (➔Signalmechanismus von OS9 wird genutzt !! Vorsicht es darf kein sigmask(1) gesetzt sein!!)**
  2. **Lesen von einer leeren Pipe ►Empfänger erhält EOF-Fehler**
  3. **Lesen wenn zu wenig Daten da, aber kein! EOF im Puffer ►Empfänger blockiert und wartet auf nächste Zeichen**
  4. **Lesen wenn zu wenig Daten da, aber EOF in der Pipe ►Empfänger erhält alle verfügbaren Daten und Anzahl der tatsächlich gelesenen Daten < Anforderung zurück aber kein EOF-Fehler.**
  5. **Öffnen einer nicht existierenden Pipe ►Fehlermeldung**
  6. **Alle Benutzer schließen den Zugriff zu einer named-Pipe ►wenn noch Daten in der Pipe, dann bleibt sie bestehen, ansonsten wird sie gelöscht.**

- **Programmgesteuertes Anlegen/Öffnen und Schließen einer named-Pipe: *_os_create(),_os_open(),_os_close()***
  - ***Create:***

```
#include <modes.h> #include <types.h>
path_id pd;
u_int32 size = 1000;  /* Maximale Pipegröße statt 128 Byte nun 1000 ►S_ISIZE zeigt im create an,
                         dass size als Parameter ausgewertet werden soll!! */
err = _os_create ("/pipe/meine_pipe", S_ISIZE| S_IREAD | S_IWRITE, &pd, FAP_READ|FAP_WRITE, size);
if (err) {
            printf("Cannot open Pipe Error:%d\n",(int)err);
            _os_exit(err);
        }
```

  - ***Open: err = _os_open ("/pipe/meine_pipe",S_IREAD | S_IWRITE,&pd);***
  - ***Close: err = _os_close (pd);***

- **Programmgesteuertes Schreiben, Lesen und Statusabfragen einer named-Pipe:**
  _os_read(),_os_write(), _os_gs_ready ():

  - ***Read: error_code = _os_read (path_id pd, void * buffer, u_int32 * count);***

    ►count enthält beim Aufruf die Wunschanzahl und bei der Rückkehr die tatsächlich gelesene Anzahl der Bytes ►EOF-Error erhält man erst bei einem Aufruf wenn kein Byte da ist.

  - ***Write: error_code = _os_write (path_id, void * buffer, u_int32 *count);***

    ►count enthält beim Aufruf die Wunschanzahl und bei der Rückkehr die tatsächlich geschriebene Anzahl der Bytes

Lese- Statusabfrage:

- ***#include <sg_codes.h>***
  #include <errno.h>
  error_code = _os_gs_ready (path_id pd, u_int32 *size);

  ►Der Aufruf prüft wieviel Bytes noch zu lesen da sind und gibt die Anzahl in size zurück. Gibt es keine so wird der Fehler EOS_NOTRDY = 246 zurückgegeben und size enthält keinen sinnvollen Wert.

Warten auf neue Zeichen mit Timeout- Abbruchmöglichkeit:

- ***Idee: Man lässt sich vom Treiber ein Signal <n> schicken wenn ein Zeichen gekommen ist.***
- ***Wichtig: Eine eigene Signalintercept-Routine muss installiert sein***

Beispiel:

```
u_int32 time = 1000;                                  /* Timeout 1000 TICS */
_os_sigmask(1);                                       /*sperre gegen Signalempfang */
_os_ss_sendsig(path_id pd, (signal_code) 350);    /* Sende  mir Signal 350 wenn auf path_id pd neue Daten da
                                                         sind */
_os_sleep(&time, &sig);                               /* Schlafe maximal time TICS , werde geweckt entweder nach
                                                         Timeout oder vorher durch Signalempfang 350 weil ein
                                                         Datum da ist */
_os_ss_relea(path_id pd);                             /* In jedem Fall delete den Signalsendeauftrag, egal ob
                                                         Timeout oder das Signal geschickt wurde */
if ( (time!=0) || (sig==350) )
        {/* nun ist ein Zeichen da zum lesen auf path_id pd*/}
else
        {/* Timeout!! */ }
```