KAPITEL 2
TASK-KOMMUNIKATION

2.1 Gegenseitiger Task-Ausschluß

In einem Multitasking-System ist eine sinnvolle Parallelität nur möglich, wenn die Anwender-Tasks eng miteinander verbunden sind. Solche Verbindungen treten aber nur gelegentlich auf und zwar immer dann, wenn es zwischen den Anwender-Tasks zum **Datenaustausch** kommt. Dies kann z. B. über eine **globale** Variable **n** erfolgen, die in einem gemeinsamen Datensegment definiert ist.

Wenn n von mehreren Anwender-Tasks als **Zähler** benutzt wird, dann führen alle Anwender-Tasks die Anweisung **n++**; aus, die z. B. durch die folgenden drei Assembler-Befehle realisiert ist:

```
MOV  AL,n     ;AL=globale Variable n
ADD  AL,1     ;AL=n+1
MOV  n,AL     ;globale Variable n=n+1
```

Aus der Befehlsfolge ist zu erkennen, daß jede Task **zweimal** auf die globale Variable n zugreift. Betrachten wir dazu ein Szenarium, wie es im folgenden Bild illustriert ist:

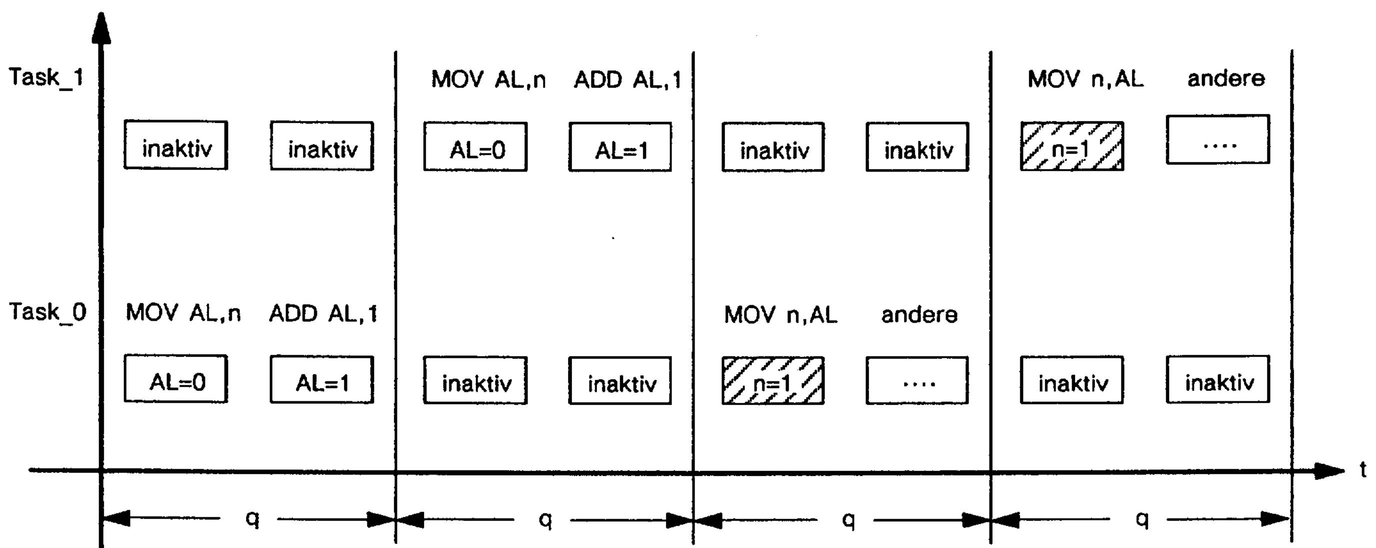
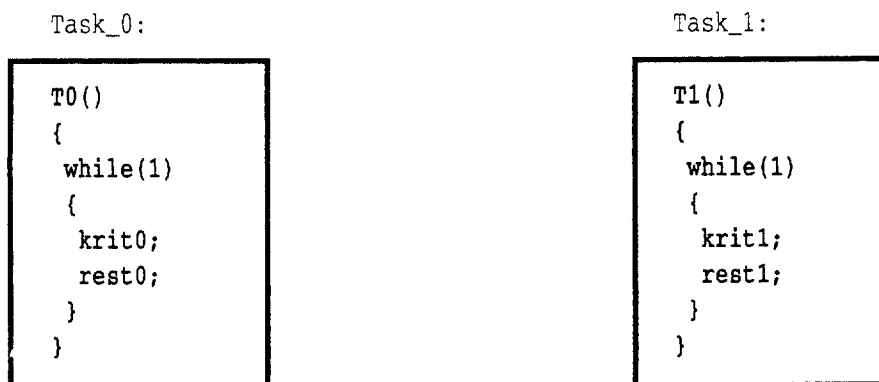


Bild 113: Überlappende Aktivitäten von Task_0 und Task_1

Die Variable n habe den ursprünglichen Wert 0, und die Task_0 beginne ihre Aktivitäten. Sie führt die Befehle MOV AL,n und ADD AL,1 aus, und das AL-Register der CPU nimmt somit den Wert 1 an. Danach wird sie verdrängt, und Task_1 beginnt ihre Aktivitäten. Sie führt ebenfalls die Befehle MOV AL,n und ADD AL,1 aus, und das AL-Register erhält nochmals den Wert 1. Anschließend wird Task_1 verdrängt, und Task_0 setzt ihre Aktivitäten fort. Sie speichert mit MOV n,AL den Wert 1 in n und führt danach **andere** nicht zu n++ gehörige Befehle aus. Schließ-

lich wird auch sie wieder verdrängt, und Task_1 führt ihr MOV n,AL aus und speichert gleichfalls den Wert 1 in n.
Die Variable n erhält also den endgültigen Wert 1, und dies, obwohl n zweimal inkrementiert worden ist. Dieses Phänomen kommt deswegen zustande, weil sich die Aktivitäten A0 von Task_0 mit den Aktivitäten A1 von Task_1 überlappen. So müssen Vorkehrungen getroffen werden, um die Aktivitäten der beiden Tasks **gegenseitig auszuschließen.** Mit anderen Worten, es muß sichergestellt sein, daß die Ausführung von n++; nur einer Task gelingt. Die **andere** Task bleibt währenddessen **blockiert** und wartet solange, bis die Gewinner-Task ihre Aktivitäten beendet hat.

Das Problem des gegenseitigen Ausschlusses kann allgemein folgendermaßen formalisiert werden:



In einer Endlosschleife führen Task_0 und Task_1 zwei unterschiedliche Klassen von Programm-Regionen aus:

1, Die **kritischen Regionen** krit0 und krit1 **mit** Zugriffen auf die gemeinsamen Ressourcen und
2, die **nicht**-kritischen Regionen rest0 und rest1 **ohne** Zugriff auf die gemeinsamen Ressourcen.

So kann krit0 und krit1 z. B. durch n++ ersetzt werden und rest0 und rest1 durch andere Befehle. Der Datenaustausch zwischen den beiden Tasks findet also immer **innerhalb** ihrer kritischen Regionen statt. Dabei dürfen sich diese Regionen nicht gegenseitig stören und müssen außerdem sicher sein, daß sie zu jeder Zeit aktuelle Daten vorfinden. Daher lautet die Forderung: krit0 und krit1 dürfen nur unter gegenseitigem Ausschluß ausgeführt werden.

2.2 Semaphore

Der verschachtelte Ablauf der kritischen Programm-Regionen kann durch sogenannte **Semaphore** (deutsch: Zeichenträger) verhindert werden, die von dem holländischen Informatiker Dijkstra eingeführt wurden. Sie sind einfach zu implementieren und ausreichend mächtig, um Tasks zu synchronisieren.
Ein Semaphor s ist eine Integer-Variable, die nur **nicht-negative** ganze Zahlen annehmen kann. Nehmen sie nur die Werte 0 und 1 an, dann heißen sie **binäre** Semaphore. Nehmen sie auch beliebige nicht-negative ganze Zahlen an, dann heißen sie **allgemeine** Semaphore. So kann mit Hilfe von Semaphoren das Problem des gegenseitigen Ausschlusses folgendermaßen gelöst werden:

Task_0:

```
T0()
{
 while(1)
 {
  P(s);
  krit0;
  V(s);
  rest0;
 }
}
```

Task_1:

```
T1()
{
 while(1)
 {
  P(s);
  krit1;
  V(s);
  rest1;
 }
}
```

Wenn Tasks in ihre kritischen Regionen **eintreten** und wenn sie diese wieder **verlassen**, dann müssen sie einige Befehle ausführen, die wir **Protokolle** nennen. Diese sollen sicherstellen, daß unabhängig von der Priorität einer Task ihre kritische Region unter Ausschluß anderer kritischer Regionen abläuft.
Den Eintritt in eine kritische Region **erlaubt** oder **verhindert** die Operation **P(s)** (**P**assieren des «Tors» zur kritischen Region), die wie folgt definiert ist:

P(s):

```
if(s > 0)
   s--;
else
   suspendieren(warten_auf_s);
```

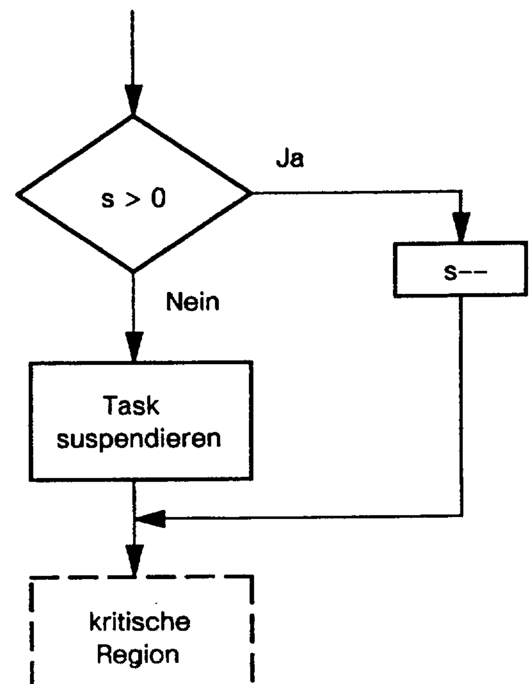


Bild 114: Operation P(s)

P(s) benutzt das gemeinsame binäre Semaphore s, das **anfangs** auf **1** gesetzt ist. Jede Task, die P(s) ausführt, prüft als erstes s.

- Bei s = **1** ist der Weg in die kritische Region **frei** und
- bei s = **0** ist der Weg versperrt.

Stellt eine Task s = 1 fest, schließt sie mit **s--**; (s = 0) das «Tor» und zeigt damit jeder anderen Task an, daß sie kurz davor ist, in ihre kritische Region einzutreten oder sich bereits in ihr befindet. So hat s nur dann den Wert 0, wenn sich genau **eine** Task in ihrem kritischen Abschnitt befindet.

Einer anderen Task, die s = 0 feststellt, bleibt daher nichts anderes übrig als zu warten. Sie wird **suspendiert**, bevor sie ihren kritischen Abschnitt betreten kann, und geht somit vom Zustand **laufend** über in den Zustand **warten_auf_s**.
Wenn eine Task ihren kritischen Abschnitt **verläßt**, führt sie die Operation **V(s)** aus (Verlassen der kritischen Region), die wie folgt definiert ist:

V(s):

```
if(andere_task == ready)
   s++;
else
   aufwecken(andere_task,warten_auf_s);
```

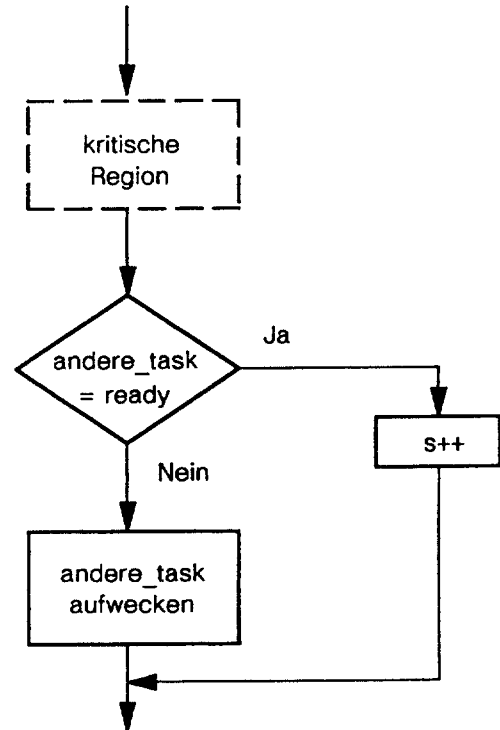


Bild 115: Operation V(s)

Jede Task, die ein V(s) ausführt, prüft als erstes, ob sich eine andere Task

- im **ready**-Zustand befindet oder
- im Zustand **warten_auf_s**.

Befindet sich die andere Task im ready-Zustand (sie ist in einer Task-Queue eingekettet), dann öffnet die aktuelle Task mit **s++**; (s = 1) das «Tor» zur kritischen Region und die andere Task kann sie betreten.

Befindet sich dagegen die andere Task im Wartezustand (sie ist in keiner Task-Queue eingekettet), dann «**weckt**» die aktuelle Task die wartende Task auf und bringt sie vom Zustand **warten_auf_s** in den Zustand **ready**.
Die so aus ihrem «Schlaf gerissene» Task führt daraufhin nach der Zuteilung durch das Scheduler/Dispatcher-Gespann ihre Arbeit fort und betritt ihre kritische Region.
Zu bemerken ist, daß die V(s) ausführende Task s **nicht** berührt hat. So bleibt s = 0, und damit das «Tor» zur kritischen Region geschlossen.
Wenn jetzt die augenblicklich aktive Task ihre kritische Region mit V(s) verläßt und dabei festellt, daß sich die andere Task im ready-Zustand befindet, dann öffnet sie mit **s++**; (s = 1) das «Tor», und die vorherige Task kann wieder die kritische Region betreten. So werfen sich die Tasks den

Schlüssel (s=1) zur kritischen Region immer wechselseitig zu, so daß sich keine Task gleichzeitig mit einer anderen in der kritischen Region aufhalten kann.
Mit jeder P(s)-Operation wird s dekrementiert, und mit jeder V(s)-Operation inkrementiert. Hat s **anfangs** den Wert **0**, dann werden alle Tasks von P(s) **suspendiert**, und keine führt jemals ein V(s) aus. So bleibt s = 0 und die Tasks warten ewig, sie sind **ausgesperrt**. Eine solche Situation wird als **Verklemmung** (deadlock) bezeichnet, die nur verhindert werden kann, wenn s anfangs 1 ist.
Wenn eine Task gerade ein P(s) oder V(s) ausführt, dann müssen andere Tasks davon abgehalten werden, gleichzeitig das Semaphor zu beeinflussen. So sind die Anweisungen s--; in P(s) und s++; in V(s) ebenfalls kritische Regionen, die nicht unterbrochen werden dürfen. Dies bedeutet, daß P(s) und V(s) als **unteilbare** (atomare) Operationen implementiert sein müssen.

Daher kann jede **P(s)**-Operation ersetzt werden durch:

```
Unterbrechung sperren;
if(s > 0)
   s--;
else
   suspendieren(warten_auf_s);
Unterbrechung zulassen;
```

und jede **V(s)**-Operation kann ersetzt werden durch:

```
Unterbrechung sperren;
if(andere_task == ready)
   s++;
else
   aufwecken(andere_task,warten_auf_s);
Unterbrechung zulassen;
```

Damit P(s) und V(s) von allen Anwender-Tasks ausgeführt werden können, sind sie als Betriebssystem-Funktionen **P_s** und **V_s** implementiert, die über folgende Call-Gate-Deskriptoren erreichbar sind:

- `KN_P_s(void);`
- `KN_V_s(void);`

Wenn die beiden Anwender-Tasks Task_0 und Task_1 die **gemeinsame** Variable **n** als Zähler benutzen, dann realisieren sie den gegenseitigen Ausschluß auf folgende Weise:

```
extern void far KN_write_io(short int,char *);
extern void far KN_P_s(void);
extern void far KN_V_s(void);
extern char far n;
static char b;

task_0_proc()
{
 while(1)

 {
  short int port_0=0x300;

  KN_P_s();                   /* P(s)  */
  b = n++;                    /* krit0 */
  KN_V_s();                   /* V(s)  */
  KN_write_io(port_0,&b); /* rest0 */
 }
}
```

Bild 116: Anwender-Modul der Task_0 (Datei task_0.c)

```
extern void far KN_write_io(short int,char *);
extern void far KN_P_s(void);
extern void far KN_V_s(void);.
extern char far n;
static char b;

task_1_proc()
{
 while(1)

 {
  short int port_1=0x301;

  KN_P_s();                   /* P(s)  */
  b = n++;                    /* krit1 */
  KN_V_s();                   /* V(s)  */
  KN_write_io(port_1,&b); /* rest1 */
 }
}
```

Bild 117: Anwender-Modul der Task_1 (Datei task_1.c)

Die Variable n ist **extern** definiert, und zwar im **DATA**-Segment des Assembler-Moduls **common**. Siehe Bild 118:

```
          NAME common

  PUBLIC n

  DATA SEGMENT RW PUBLIC

  n    DB    0

  DATA ENDS
          END
```

Bild 118: Definition der gemeinsamen Variable n (Datei common.src)

Für beide Tasks ist n eine **FAR**-Variable, auf die sie uneingeschränktes Zugriffsrecht haben müssen. Daher liegt das DATA-Segment von n im **User**-Bereich (Privileg-Ebene 3) und wird von einem Deskriptor spezifiziert, der von jeder Task erreichbar in der GDT enthalten ist. Siehe Bild 119:

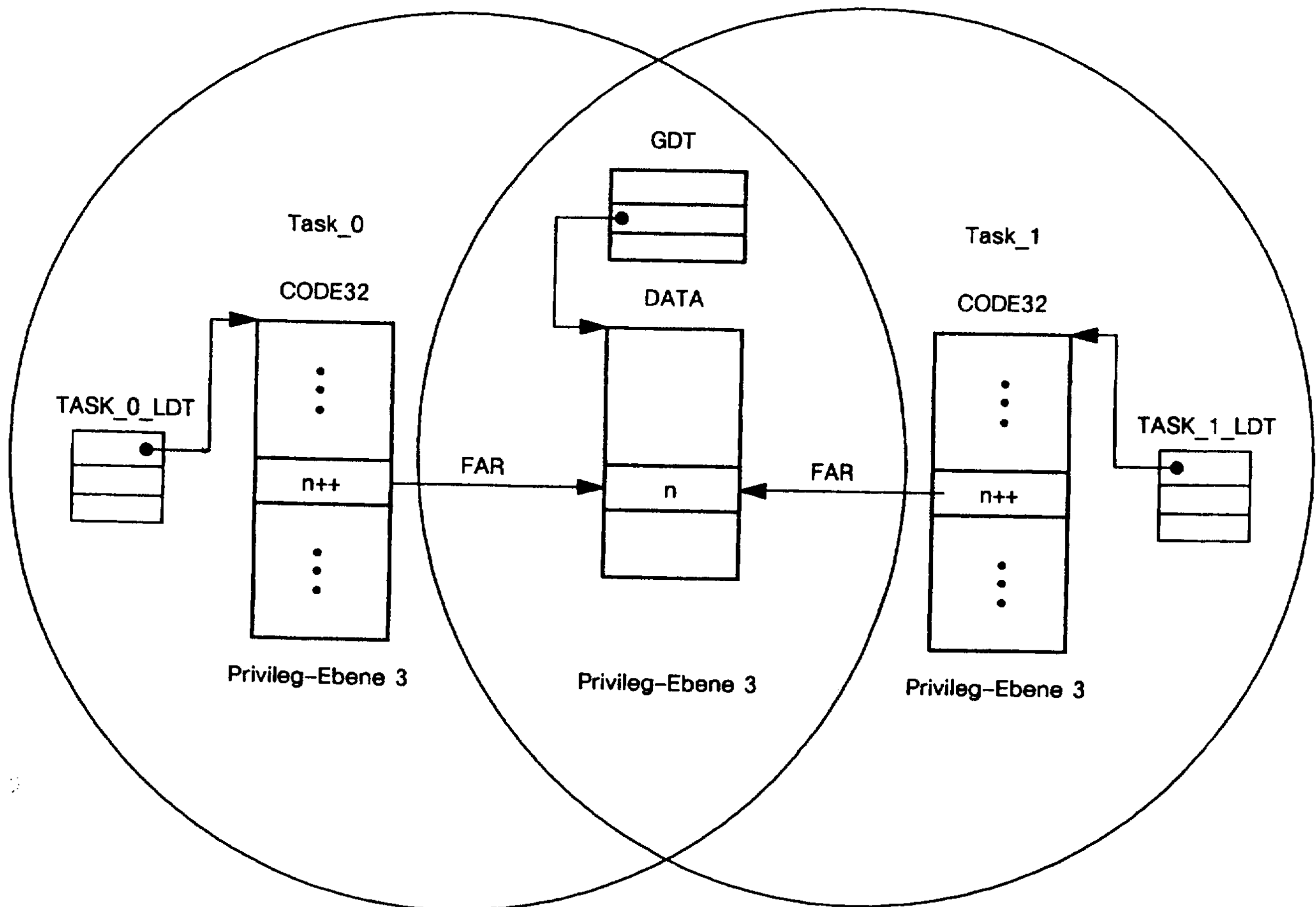


Bild 119: Die FAR-Variable n kann von jeder Task erreicht werden

In krit0 und krit1 der beiden Anwender-Tasks wird der augenblickliche Zählerstand von n zunächst in der lokalen (static) Variablen **b** zwischengespeichert, und danach wird b von rest0 bzw. rest1 zur Anzeige gebracht.
Was würde passieren, wenn die Tasks die gemeinsame Variable n **nicht** über ihre Hilfsvariablen b, sondern unmittelbar ausgeben würden? Dann würden die Anweisungen

- `KN_write_io(port_0,&n)` und `KN_write_io(port_1,&n)`

zu kritischen Regionen werden, die ebenfalls geschützt werden müßten.
Wie nun im einzelnen die Anwender-Tasks mit den Betriebssystem-Funktionen **P_s** und **V_s** korrespondieren, zeigt Bild 120:

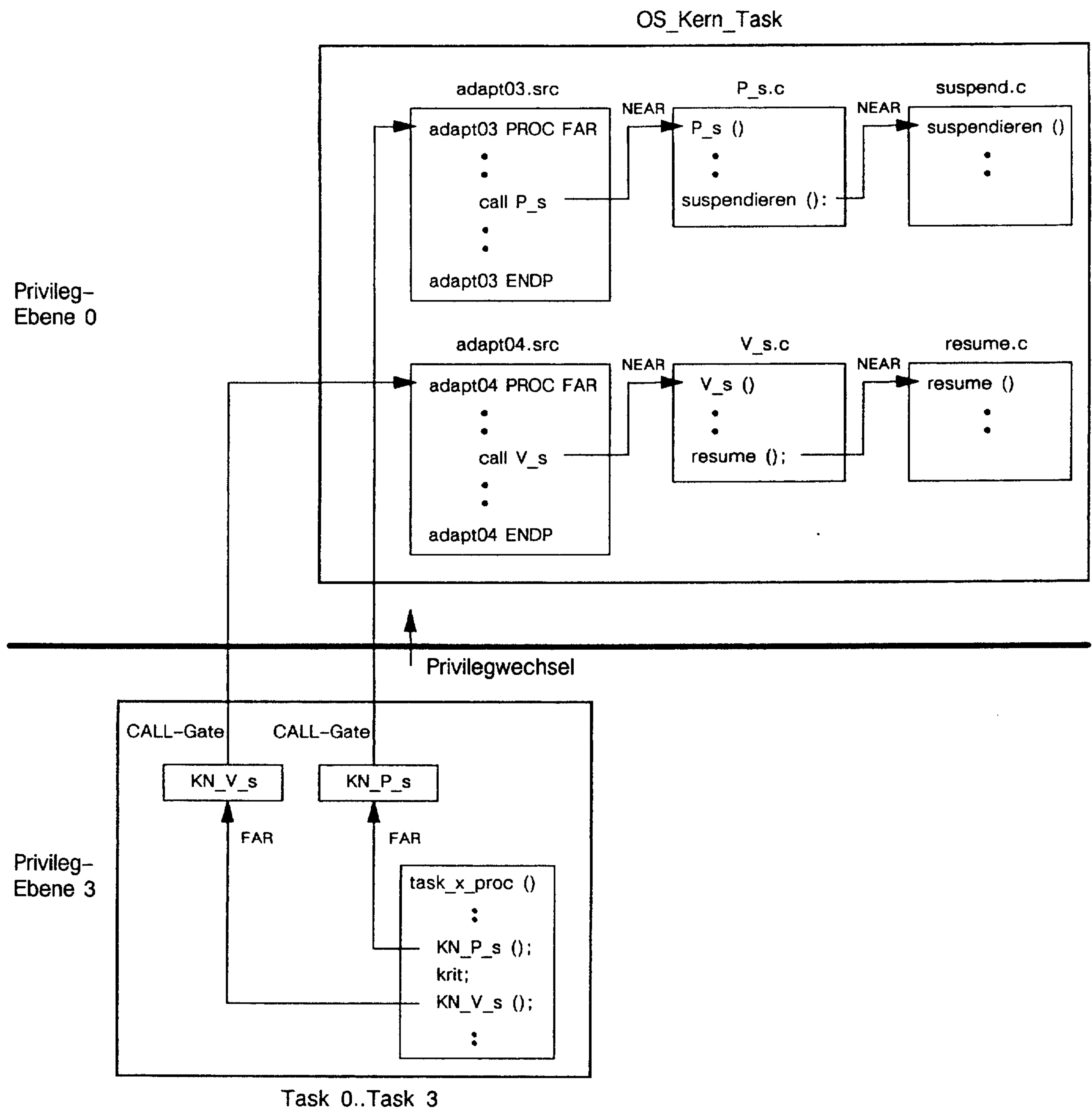


Bild 120: Aufruf der Betriebssystem-Funktionen P_s und V_s

Zwischen den Anwender-Tasks und den Betriebssystem-Funktionen liegen die Assembler-Adapter **adapt03** und **adapt04.** Weil weder P_s noch V_s unterbrochen werden darf, sperren die Adapter als erstes mit CLI potentielle Interruptanforderungen. Sie laden anschließend, wie üblich, den DS-Selektor der aktiven Anwender-Task um und rufen danach die Betriebssytem-Funktion auf. Erfolgt die Rückkehr zur Anwender-Task, dann lassen die Adapter Programm-Unterbrechungen mit STI wieder zu.
Wenn P_s feststellt, daß wegen s=0 der Weg zur kritischen Region b = n++; versperrt ist, ruft sie die Betriebssystem-Funktion **suspendieren** auf.
Ähnliches gilt für V_s. Wenn sie feststellt, daß die andere Task auf das Semaphor s wartet, dann ruft sie die Betriebssystem-Funktion **resume** auf.

Die Assembler-Programme der beiden Adapter sehen wie folgt aus:

```
        NAME adapt03

PUBLIC adapt03
EXTRN P_s:NEAR

DATA SEGMENT RW PUBLIC
     DD  ?
DATA ENDS

CODE32 SEGMENT ER PUBLIC

adapt03 PROC FAR WC(0)
        cli                         ;Interrupt sperren
        push ebp                    ;Prolog
        mov  ebp,esp
        push ds                     ;DS-Selektor der Anwender-Task retten
        mov  ax,data                ;DS-Selektor/Deskriptor der OS-Kern-Task
        mov  ds,ax                  ;laden
        call P_s                    ;Aufruf der Beriebssystem-Funktion
        pop  ds                     ;DS-Selektor/Deskriptor der Anwender-Task
        pop  ebp                    ;Epilog
        sti                         ;Interrupt freigeben
        ret                         ;Zurueck zur Anwender-Task
adapt03 ENDP

CODE32 ENDS
       END
```

Bild 121: Adapter adapt03 für Betriebssystem-Funktion P_s (Datei adapt03.src)

```
PUBLIC adapt04
EXTRN V_s:NEAR

DATA SEGMENT RW PUBLIC
     DD  ?
DATA ENDS

CODE32 SEGMENT ER PUBLIC

adapt04 PROC FAR WC(0)
        cli                    ;Interrupt sperren
        push ebp               ;Prolog
        mov  ebp,esp
        push ds                ;DS-Selektor der Anwender-Task retten
        mov  ax,data           ;DS-Selektor/Deskriptor der OS-Kern-Task
        mov  ds,ax             ;laden
        call V_s               ;Aufruf der Beriebssystem-Funktion
        pop  ds                ;DS-Selektor/Deskriptor der Anwender-Task
        pop  ebp               ;Epilog
        sti                    ;Interrupt freigeben
        ret                    ;Zurueck zur Anwender-Task
adapt04 ENDP

CODE32 ENDS
       END
```

Bild 122: Adapter adapt04 für Betriebssystem-Funktion V_s (Datei adapt04.src)

Betrachten wir jetzt die Betriebssystem-Funktion P_s:

```
#define warten_auf_s 6

extern char s;

void P_s(void)
{
 if(s > 0)
    s --;
 else
    suspendieren(warten_auf_s);
}
```

Bild 123: Betriebssystem-Funktion P_s (Datei P_s.c)

P_s prüft als erstes s; wenn sie s=1 feststellt, dann ist der Weg zur kritischen Region für die augenblicklich aktive Anwender-Task frei. Mit s--; versperrt sie daraufhin den Weg für die andere Task und veranlaßt über adapt03 die Rückkehr in die Privileg-Ebene 3. Dort betritt dann die aktive Anwender-Task ihre kritische Region und führt b = n++; aus.
Wenn aber P_s festellt, daß s=0 ist, dann befindet sich die andere Task in ihrer kritischen Region. P_s **suspendiert** daraufhin die augenblicklich aktive Task und veranlaßt einen Taskwechsel. Zu diesem Zweck benutzt sie die folgende Betriebssystem-Funktion **suspendieren:**

```
#define prioritaet_max 4
#define task_max       4
#include<struct.h>

extern task_queue_kopf_struct task_queue_kopf[prioritaet_max];
extern task_cb_struct task_cb[task_max];
extern char aktuell_task_id;

void suspendieren(char neu_task_status)
{
 ausketten(&task_queue_kopf[task_cb[aktuell_task_id].task_prioritaet]);
 task_cb[aktuell_task_id].task_status=neu_task_status;
 task_wechsel();
}
```

Bild 124: Betriebssystem-Funktion suspendieren (Datei suspend.c)

Wie zu sehen ist, wird suspendieren mit einem **einzigen** Parameter aufgerufen. Er bestimmt den künftigen Zustand der zu suspendierenden Task und lautet in unserem Fall **warten_auf_s**. Dieser Zustand ist z. B. durch die Identifikationsziffer **6** gekennzeichnet.
Die Funktion beginnt ihre Arbeit, indem sie als erstes den **Task-Control-Block** der betreffenden Task aus der Task-Queue auskettet. Sie ruft zu diesem Zweck die Betriebssystem-Funktion **ausketten** auf und übergibt ihr als Parameter die Adresse des richtigen **Task-Queue-Kopf's**. Anschließend initialisiert sie mit

- `task_cb[aktuell_task_id].task_status = neu_task_status;`

das task_status-Feld des ausgeketteten Task-Control-Blocks mit dem **neuen** Task-Zustand (neu_task_status), also mit warten_auf s=6. Wenn Task_0 beispielsweise an der Prioritätsstufe 1 suspendiert wird, dann ist aktuell_task_id=0, und **suspendieren** führt die Anweisung task_cb[0].task_status=6; aus. So ergibt sich für den betreffenden Task-Control-Block das folgende Layout:

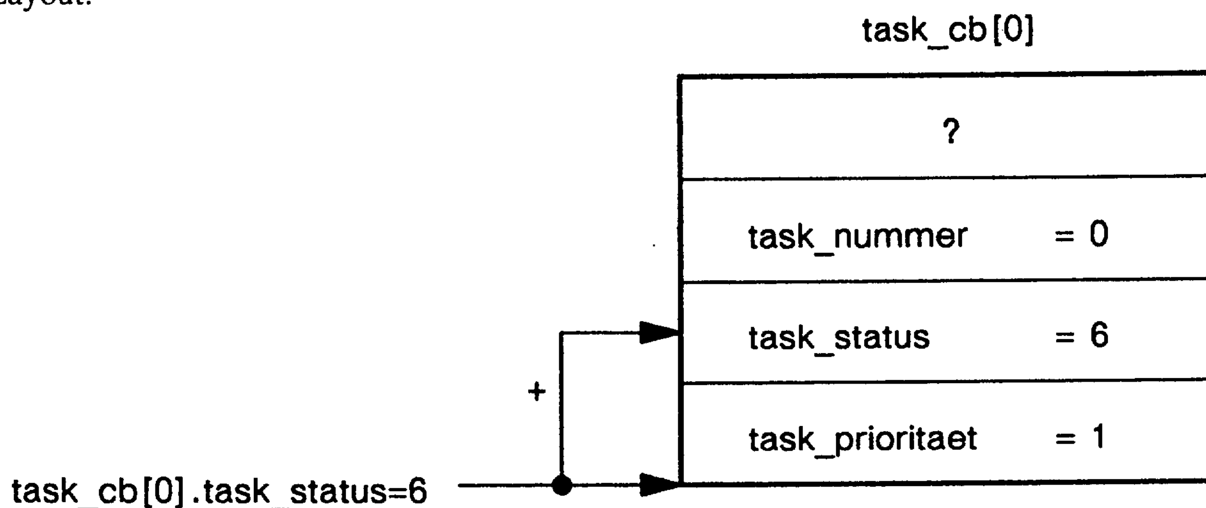


Bild 125: Task-Control-Block der suspendierten Task_0

Dabei zeigt task_status=6 an, daß sich die Task_0 im Zustand warten_auf_s befindet.
Trotz dieses Zustands ist Task_0 noch immer im Besitz der CPU. Daher veranlaßt suspendieren

noch den notwendigen Taskwechsel, so daß danach die andere Task ihre kritische Region betreten kann.
Betrachten wir als nächstes die Betriebssystem-Funktion **V_s**:

```
#include<struct.h>
#define task_max            4
#define ready               1
#define warten_auf_s        6

extern task_cb_struct task_cb[task_max];
extern char s;
extern char aktuell_task_id;
extern char andere_task_id;

void V_s(void)
{
 switch(aktuell_task_id)
 {
  case 0:  andere_task_id=1;
           break;
  case 1:  andere_task_id=0;
           break;
  default: break;
 }

 if
   (task_cb[andere_task_id].task_status == ready)
    s ++;
 else
    resume(andere_task_id,warten_auf_s);
}
```

Bild 126: Betriebssystem-Funktion V_s (Datei V_s.c)

Wenn eine Anwender-Task V_s ausführt, dann stellt V_s als erstes fest, ob sich die andere Task im **ready**-Zustand befindet oder nicht. Zu diesem Zweck überprüft sie das **task_status**-Feld im Task-Control-Block der anderen Task, wofür sie deren Identifikationsnummer benötigt.
Wenn sich nur Task_0 und Task_1 die gemeinsame Variable n teilen, dann ist die Identifikationsnummer entweder 0 oder 1 und kann aus **aktuell_task_id** auf einfache Weise ermittelt werden:

- Führt Task_0 V_s aus, dann ist aktuell_task_id=0 und die Nummer der anderen Task ist 1.
- Führt Task_1 V_s aus, dann ist aktuell_task_id=1 und die Nummer der anderen Task ist 0.

In V_s ist dieser Algorithmus mit Hilfe der **switch**-Anweisung realisiert. Sie ermittelt die Nummer der anderen Task und speichert sie in der char-Variablen **andere_task_id**, die extern in der Betriebssystem-Funktion init definiert ist.

- Führt Task_0 V_s aus, dann prüft task_cb[1].task_status==ready den Zustand der Task_1.
- Führt Task_1 V_s aus, dann prüft task_cb[0].task_status==ready den Zustand der Task_0.

Stellt V_s fest, daß die andere Task ready ist, dann öffnet sie mit s++; das «Tor» zur kritischen

Region. Im anderen Fall «weckt» sie die suspendierte Task auf und veranlaßt einen Taskwechsel. Sie benutzt dafür die folgende Betriebssystem-Funktion **resume**:

```
#define prioritaet_max      4
#define ready               1
#define task_max            4
#include<struct.h>

extern task_queue_kopf_struct task_queue_kopf[prioritaet_max];
extern task_cb_struct task_cb[task_max];

void resume(char task_id,char neu_task_status)
{
 task_queue_kopf_struct *task_queue_kopf_adr;
 task_cb_struct *neu_task_cb_adr;

 if(task_cb[task_id].task_status == neu_task_status)
    {
     task_cb[task_id].task_status = ready;
     task_queue_kopf_adr=&task_queue_kopf[task_cb[task_id].task_prioritaet];
     neu_task_cb_adr=&task_cb[task_id];
     einketten(task_queue_kopf_adr,neu_task_cb_adr);

     task_wechsel();
    }
}
```

Bild 127: Betriebssystem-Funktion resume (Datei resume.c)

Wie zu sehen ist, wird resume mit **zwei** Parametern aufgerufen. Dabei bekommt der linke Parameter task_id die Nummer der aufzuweckenden Task (andere_task_id = 0 oder 1), und der rechte Parameter neu_task_status bekommt die Identifikationsziffer des augenblicklichen Wartezustands (6 für warten_auf_s).
Die Funktion beginnt ihre Arbeit, indem sie als erstes mit

- `if(task_cb[task_id].task_status == neu_task_status)`

überprüft, ob sich die aufzuweckende Task im **gesuchten** Zustand befindet (hier im Zustand warten_auf_s) oder nicht. Siehe Bild 128.

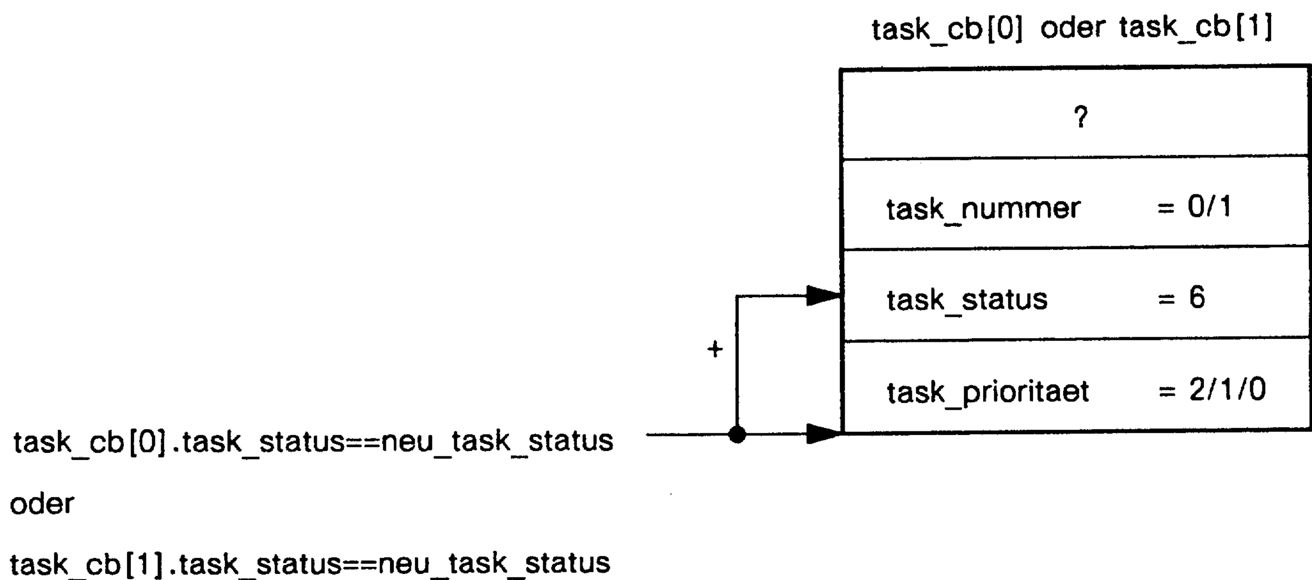


Bild 128: Zustand der aufzuweckende Task feststellen

Wenn **nein**, erfolgt über adapt04 die Rückkehr in die Privileg-Ebene 3.
Wenn **ja**, dann initialisiert sie mit

- `task_cb[task_id].task_status = ready;`

das task_status-Feld des betreffenden Task-Control-Blocks mit **1** und bringt so die Anwender-Task vom Wartezustand in den Zustand ready. Siehe Bild 129.

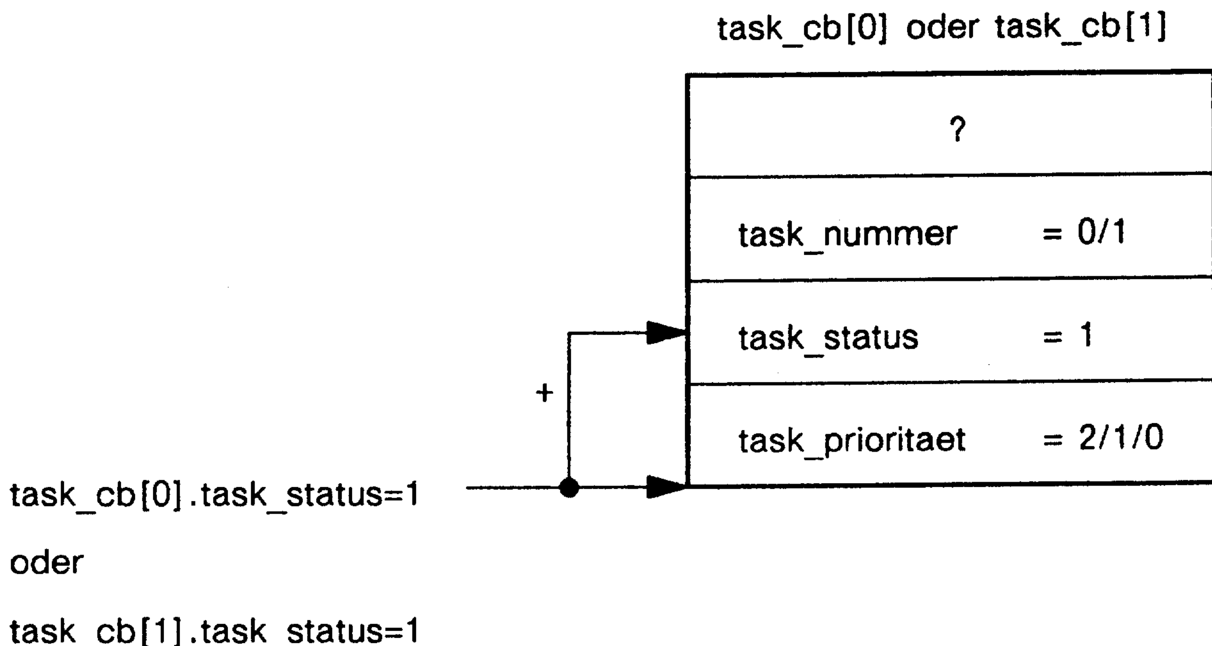


Bild 129: Die aufzuweckende Task bekommt den Zustand ready

Nachdem dies geschehen ist, kettet resume den Task-Control-Block an der augenblicklichen Prioritätsstufe (2,1 oder 0) in die Task-Queue ein und ruft zu diesem Zweck die Betriebssytem-Funktion **einketten** auf. Weil aber die weckende Task noch immer im Besitz der CPU ist, veranlaßt resume noch den notwendigen Taskwechsel, so daß danach die aufgeweckte Task ebenfalls ihre kritische Region betreten kann.

2.3 Gegenseitiger Ausschluß bei mehreren Tasks

Auch bei mehreren Anwender-Tasks kann das Problem des gegenseitigen Auschlusses mit P(s)- und V(s)-Operationen gelöst werden. Besteht das Multitasking-System z. B. aus vier Anwender-Tasks, dann lautet die Lösung:

```
Task_0:         Task_1:         Task_2:         Task_3:

t0()            t1()            t2()            t3()
{               {               {               {
 while(1)        while(1)        while(1)        while(1)
 {               {               {               {
  P(s);           P(s);           P(s);           P(s);
  krit0;          krit1;          krit2;          krit3;
  V(s);           V(s);           V(s);           V(s);
  rest0;          rest1;          rest2;          rest3;
 }               }               }               }
}               }               }               }
```

Betrachten wir hierzu folgendes Szenarium. Es sei s=1.

1. Task_0 besitzt die CPU, führt P(s) aus, setzt s=0 und betritt krit0.
2. krit0 wird unterbrochen, Task_0 verläßt die CPU, Task_1 nimmt sie in Besitz und führt P(s) aus. Weil s=0 ist, wird Task_1 suspendiert.
3. Task_2 bekommt die CPU, führt P(s) aus und wird wegen s=0 ebenfalls suspendiert.
4. Jetzt nimmt Task_3 die CPU in Besitz. Sie erfährt dasgleiche Schicksal und wird wegen s=0 suspendiert. So warten Task_1, Task_2 und Task_3 auf s=1.
5. Task_0 bekommt daraufhin wieder die CPU und beendet krit0. Danach ist die Frage zu beantworten: Welche der Tasks soll aufgeweckt werden?
6. Sie führt V(s) aus und weckt Task_1, denn sie kann nur die Nummer von Task_1 ermitteln (andere_task_id=1). Von den verbleibenden Tasks, also von Task_2 und Task_3, hat sie keinerlei Kenntnis.

Das gleiche Problem wie Task_0 hat auch Task_1. Sind Task_0, Task_2 und Task_3 suspendiert und führt Task_1 V(s) aus, dann kann sie nur die Nummer von Task_0 ermitteln (andere_task_id=0). Auch sie kennt die restlichen Tasks nicht.
So werfen sich immer nur Task_0 und Task_1 den Schlüssel zur kritischen Region zu (s=1) und «verschwören» sich auf diese Weise untereinander gegen Task_2 und Task_3.
Die Folge ist, daß Task_2 und Task_3 für immer warten und somit **ausgesperrt** bleiben.
Wie eine solche Aussperrung zu verhindern ist, zeigt der nächste Abschnitt.

2.3.1 Implementierung von Condition-Queues

Wir nehmen an, daß ein Multitasking-System nicht nur über eine einzige gemeinsame Variable verfügt, sondern über mehrere. Wir nennen sie $\mathbf{n_0,n_1..n_n}$. Jede Anwender-Task kann eine beliebige Anzahl dieser Variablen anfordern und in Beschlag nehmen.
Bild 130 zeigt hierfür ein Beispiel:

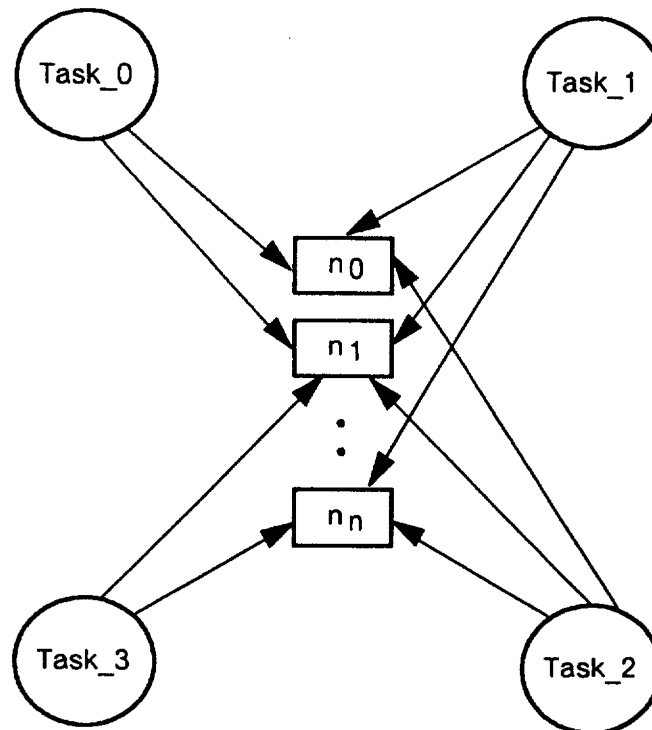


Bild 130: Multitasking-System mit mehreren gemeinsamen Variablen

Auch hier erfolgt der Zugriff auf n_0,n_1..n_n unter gegenseitigem Ausschluß, der durch weitere Semaphore wie $\mathbf{s_0}$,$\mathbf{s_1}$..$\mathbf{s_n}$ und zusätzliche P/V-Paare $P(s_0)/V(s_0)$,$P(s_1)/V(s_1)$..$P(s_n)/V(s_n)$ geregelt wird.
So können die Anwender-Tasks diese P/V-Paare z. B. folgendermaßen benutzen:

```
Task_0:          Task_1:          Task_2:          Task_3:

t0()             t1()             t2()             t3()
{                {                {                {
 while(1)         while(1)         while(1)         while(1)
 {                {                {                {
  P(s0);           P(s0);           P(s1);           P(s0);
  n0++;            n0++;            n1++;            n0++;
  V(s0);           V(s0);           V(s1);           V(s0);
  .                .                .                .
  .                .                .                .
  P(s1);           P(s1);           P(s3);           P(s3);
  n1++;            n1++;            n3++;            n3++;
  V(s1);           V(s1);           V(s3);           V(s3);
 }                }                }                }
}                }                }                }
```

Wie zu sehen ist, teilen sich hier

- Task_0, Task_1 und Task_3 die Variable n_0,
- Task_0, Task_1 und Task_2 die Variable n_1 und
- Task_2, Task_3 die Variable n_3.

Wenn die P-Operationen ihre Anwender-Tasks wegen $s_0=0$ oder $s_1=0..s_n=0$ suspendieren, dann befinden sich die betreffenden Tasks in den Wartezuständen warten_auf_s_0 oder warten_auf_s_1.. warten_auf_s_n.
Führt bei dieser Situation eine noch aktive Task eine V-Operation aus, dann muß V

1. über eine spezielle Betriebssystem-Funktion eine der «schlafenden» Tasks auswählen, d. h. ihre Identifikationsnummer ermitteln (andere_task_id=?) und sie
2. über die Betriebssystem-Funktion resume aus dem augenblicklichen Wartezustand befreien.

Steht die Identifikationsnummer zur Verfügung, dann läßt V von resume prüfen, ob sich die suspendierte Task im Zustand neu_task_status (warten_auf_s_0 oder warten_auf_s_1..warten_auf_s_n) befindet oder nicht:

- `if(task_cb[task_id].task_status == neu_task_status)` (siehe Bild 127)

Auf diese Weise **signalisiert** die V ausführende Task der suspendierten Task die **Bedingung**, unter der sie aufgeweckt wird.
Wenn der Wartegrund der suspendierten Task mit der signalisierten Bedingung übereinstimmt, dann weckt resume diese Task auf.
Im anderen Fall wirkt resume wie eine Leeranweisung, und V hinterläßt keinerlei Spuren.
Die Bedingung, unter der eine suspendierte Task aufweckbar ist, wird von ihrer P-Operation bestimmt.
So benutzen mehrere P/V-Paare eine Anzahl von Bedingungen, für die sich eine neue Variable, die sogenannte **Bedingungsvariable** definieren läßt.
Wenn **condition** diese Variable ist, dann kann sie von den folgenden zwei neuen Betriebssystem-Funktionen verarbeitet werden:

- `wait(char condition)`
- `signal(char condition)`

Eine Anwender-Task, die wait ausgeführt hat, befindet sich danach im Zustand condition und wartet darauf, daß condition eintritt.
Eine Anwender-Task, die signal ausführt, zeigt einer anderen Task an, daß condition eingetreten ist.
Mit der Einführung von wait und signal kann jetzt für jedes Semaphor, das wir allgemein **sem** nennen, ein individuelles P(sem)/V(sem)-Paar wie folgt definiert werden:

P(sem):

```
if (sem > 0)
    sem--;
else
    wait(condition);
```

V(sem):

```
if (beliebige_task == warten_auf_bedingung)
    signal(condition);
else
    sem++;
```

In P(sem) ist im Vergleich zu P(s) lediglich suspendieren durch wait ersetzt. Dagegen gestaltet sich V(sem) im Vergleich zu V(s) etwas anders. So prüft V(sem) nicht mehr, ob sich eine andere Task im ready-Zustand befindet, sondern sie stellt fest, ob eine beliebige andere Task auf **condition** wartet.

• Wenn **ja**, signalisiert sie der wartenden Task die Bedingung und weckt sie auf. Die geweckte Task betritt daraufhin die kritische Region. Dabei bleibt sem=0 und damit die kritische Region vor anderen Tasks geschützt.
• Wenn **nein**, inkrementiert sie sem und gibt wegen sem=l den Weg zur kritischen Region frei.

Sowohl wait als auch signal arbeiten eng zusammen mit einer neuen Art von Warteschlange, der sogenannten **Condition-Queue**.
Für jede Bedingung stellt das Multitasking-System eine solche Queue zur Verfügung.
Sie hat den gleichen Aufbau wie die Task-Queue und besteht aus einem **Condition-Queue-Kopf** und einer Anzahl von **Condition-Elementen**. Jeder Anwender-Task ist ein eigenes Condition-Element zugeordnet. Es enthält als wichtigsten Bestandteil die Nummer der betreffenden Task. Wenn eine Anwender-Task wait(condition) ausführt, dann wird ihr Task-Control-Block aus der Task-Queue ausgekettet und ihr Condition-Element in die Queue der spezifizierten Bedingung eingekettet. Dort ist sie entweder alleine oder sie gesellt sich zu anderen Tasks, die genauso wie sie auf die gleiche Bedingung warten.

Wenn umgekehrt eine Anwender-Task signal(condition) ausführt, dann stellt signal die erste Task in der Queue der betreffenden Bedingung fest und weckt sie auf.
Angenommen, ein Multitasking-System kennt insgesamt 29 verschiedene Bedingungen, die wir $cond_2$...$cond_{30}$ nennen. Dann ist jede dieser Bedingungen in Übereinstimmung mit der folgenden Tabelle durch eine individuelle Identifikationsziffer gekennzeichnet:

condition	Identifikationsziffer
$cond_2$	2
$cond_3$	3
•	•
•	•
•	•
$cond_{30}$	30

Benutzt man die Identifikationsziffern als Indizes, dann läßt sich ein **array** von Condition-Queue-Köpfen konstruieren, wie z. B.

condition_queue_kopf[2]...condition_queue_kopf[30].

Jeder dieser Köpfe repräsentiert eine bestimmte Bedingungsstufe, an die wait(condition) das zur augenblicklich aktiven Task gehörende Condition-Element einkettet. Das folgende Bild verdeutlicht dies an einem Beispiel:

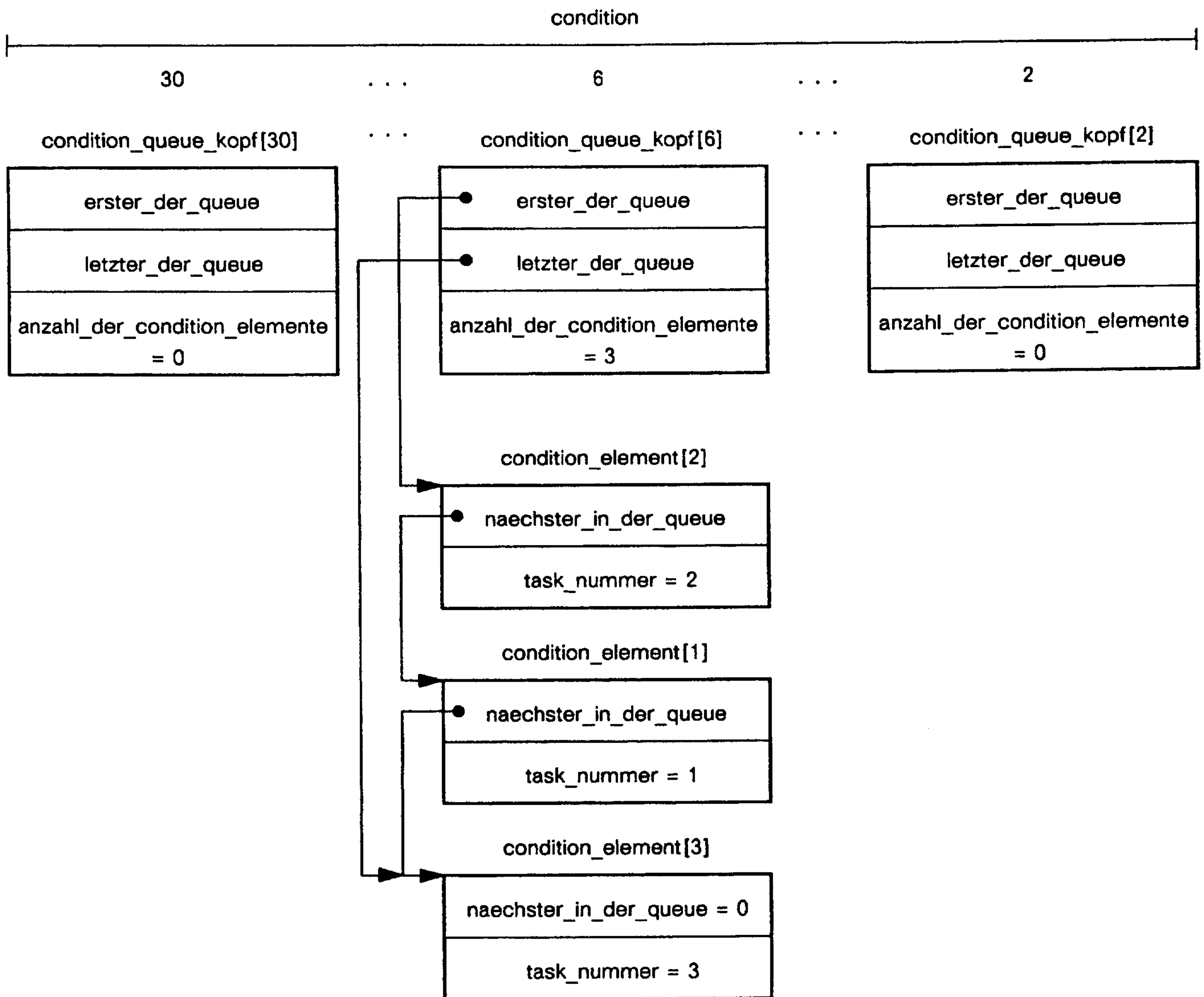


Bild 131: Beispiel einer Condition-Queue-Konfiguration

Wie zu sehen ist, enthalten die Condition-Queue-Köpfe die üblichen Zeiger **erster_der_queue** und **letzter_der_queue** zum ersten und letzten Condition-Element und den Zähler **anzahl_der_condition_elemente**. Die Condition-Elemente enthalten den üblichen Verkettungszeiger **naechster_in_der_queue** und die Variable **task_nummer**, die die suspendierte Anwender-Task identifiziert.

Im obigen Beispiel haben Task_2 (task_nummer=2), Task_1 (task_nummer=1) und Task_3 (task_nummer=3) die Betriebssystem-Funktion **wait(cond$_6$)** ausgeführt. Ihre Condition-Elemente sind daher an der Condition-Stufe **cond$_6$** mit der Identifikationsziffer **6** in die Queue eingekettet. Weil andere waits nicht ausgeführt wurden, sind die verbleibenden Condition-Queues leer.

Anzumerken bleibt, daß wait und signal die Bedingungen **cond$_0$** und **cond$_1$** mit den Identifikationsziffern **0** und **1** nicht verarbeiten. Warum? Nun, cond$_0$=0 kennzeichnet eine Task als **unbekannt** (dormant), und es ist Sache des Anwenders, sie in den ready-Zustand zu versetzen und sie so dem System bekannt zu geben. cond$_1$=1 kennzeichnet eine Task als **ready**. In diesem Zustand

wartet eine Anwender-Task in einer der Task-Queues auf die **CPU** und nicht in einer Condition-Queue auf die Erfüllung einer Bedingung.
Die Struktur der Condition-Queue, bestehend aus dem Condition-Queue-Kopf und den Condition-Elementen, läßt sich mit den folgenden zwei **neuen** Datentypen **s7** und **s8** beschreiben:

```
typedef struct s7
              {
                struct s8 *erster_der_queue;
                struct s8 *letzter_der_queue;
                short int anzahl_der_condition_elemente;
              } condition_queue_kopf_struct;

typedef struct s8
              {
                struct s8 *naechster_in_der_queue;
                char task_nummer;
              } condition_element_struct;
```

Bild 132: Definition der Datentypen condition_queue_kopf_struct und condition_element_struct (Datei struct3.h)

Beide sind umbenannt, und zwar s7 in **condition_queue_kopf_struct** und s8 in **condition_element_struct**.
Für die Initialisierung der Condition-Queue-Köpfe und Condition-Elemente ist die Betriebssystem-Funktion **init** eine passende Kandidatin. Und so ist ihr die Header-Datei **struct3.h** «einverleibt», in der die neuen Datentypen definiert sind. Siehe Bild 133:

```
#define null (void*)       0
#define task_max           4
#define prioritaet_max     4
#define condition_max     30
#include<struct.h>
#include<struct3.h>

task_queue_kopf_struct      task_queue_kopf[prioritaet_max];
task_cb_struct              task_cb[task_max];
condition_queue_kopf_struct condition_queue_kopf[condition_max];
condition_element_struct    condition_element[task_max];

char i,s=1,andere_task_id;

void init(void)
{
 for (i=0;i<prioritaet_max;i++)  /* 4 Prioritaetsstufen installieren */
     {
      task_queue_kopf[i].erster_der_queue    = null;
      task_queue_kopf[i].letzter_der_queue   = null;
      task_queue_kopf[i].anzahl_der_task_cbe = 0;
     }

 for (i=2;i<=condition_max;i++) /* 29 Condition-Queue-Koepfe initialisieren */
     {
      condition_queue_kopf[i].erster_der_queue                = null;
      condition_queue_kopf[i].letzter_der_queue               = null;
      condition_queue_kopf[i].anzahl_der_condition_elemente = 0;
     }

 for (i=0;i<task_max;i++)       /* 4 Condition-Elemente initialisieren */
     {
      condition_element[i].naechster_in_der_queue    = null;
      condition_element[i].task_nummer               = i;
     }
}
```

Bild 133: Betriebssystem-Funktion init; erste Erweiterung (Datei init.c)

In init sind neben den bereits bekannten arrays task_queue_kopf[prioritaet_max] und task_cb[task_max] **zwei** weitere vereinbart, nämlich

- condition_queue_kopf[condition_max], dessen Elemente die Struktur des Condition-Queue-Kopfs haben und
- condition_element[task_max], dessen Elemente die Struktur der Condition-Elemente haben.

In einer **for**-Schleife, deren Laufvariable i von **2 = i <= condition_max** läuft, werden insgesamt 29 Condition-Queue-Köpfe installiert und ihre Inhalte mit 0 initialisiert. So stehen am Ende 29 Conditionstufen mit jeweils leeren Queues zur Verfügung. Siehe Bild 134:

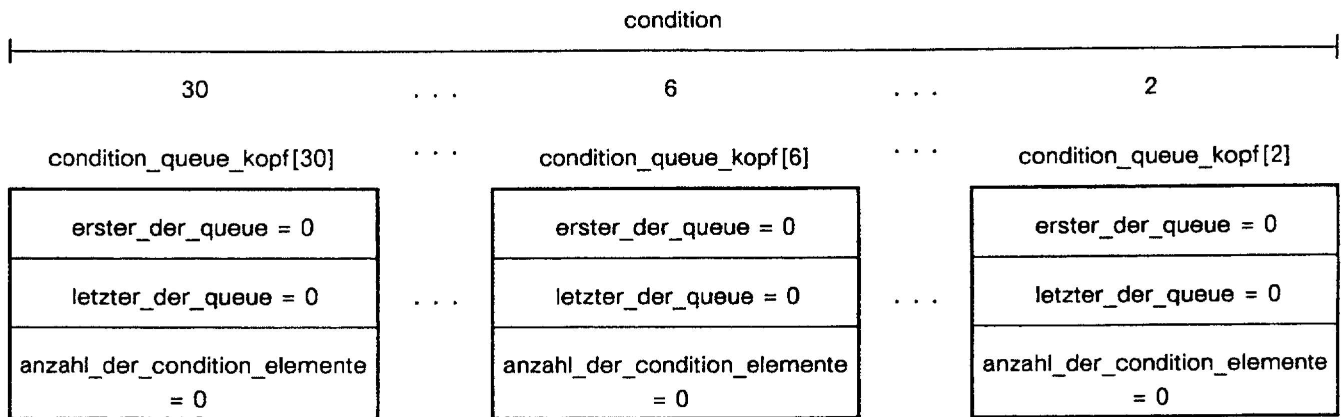


Bild 134: Bereitstellung von 29 Conditionstufen

In einer weiteren **for**-Schleife, deren Laufvariable i von **0 = i < task_max** läuft, stellt init für Task_0..Task_3 je ein Condition-Element bereit und initialisiert den Verkettungszeiger naechster_in_der_queue mit 0 und die Variable task_nummer mit der Nummer der korrespondierenden Anwender-Task. Siehe Bild 135:

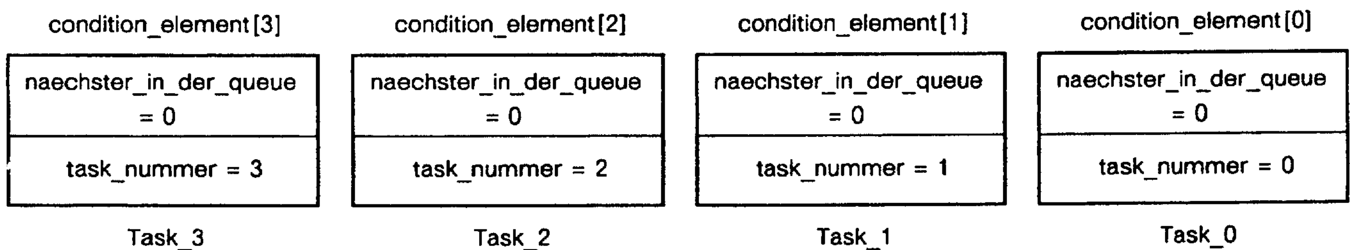


Bild 135: Bereitstellung von 4 Condition-Elementen

Betrachten wir nun die Betriebssystem-Funktion **wait**:

```
#define task_max                  4
#define condition_max            30

#include<struct3.h>

extern condition_queue_kopf_struct condition_queue_kopf[condition_max];
extern condition_element_struct    condition_element[task_max];
extern char aktuell_task_id;

void wait(char condition)
{
 condition_queue_kopf_struct *condition_queue_kopf_adr;
 condition_element_struct    *condition_element_adr;

 condition_queue_kopf_adr =
           &condition_queue_kopf[condition];
 condition_element_adr = &condition_element[aktuell_task_id];
 einketten(condition_queue_kopf_adr,condition_element_adr);
 suspendieren(condition);
}
```

Bild 136: Betriebssystem-Funktion wait (Datei condwait.c)

Wenn sie von einer Anwender-Task ausgeführt wird, dann kettet sie als erstes das Condition-Element dieser Task bei der spezifizierten condition (übergebener Parameter = $cond_x$) ein. Zu diesem Zweck ruft sie die Betriebssystem-Funktion **einketten** auf und übergibt ihr als Parameter zwei Adressen. Der linke Parameter condition_queue_kopf_adr ist vom Typ condition_queue_kopf_struct und bekommt wegen

- `condition_queue_kopf_adr = &condition_queue_kopf[condition]`

die **Adresse** eines **Condition-Kopfes.**

Der rechte Parameter condition_element_adr ist vom Typ condition_element_struct und bekommt wegen

- `condition_element_adr = &condition_element[aktuell_task_id]`

die Adresse des einzukettenden Condition-Elements. Auf welche Weise wait diese Adressen ermittelt, illustriert das folgende Bild 137 an einem Beispiel:

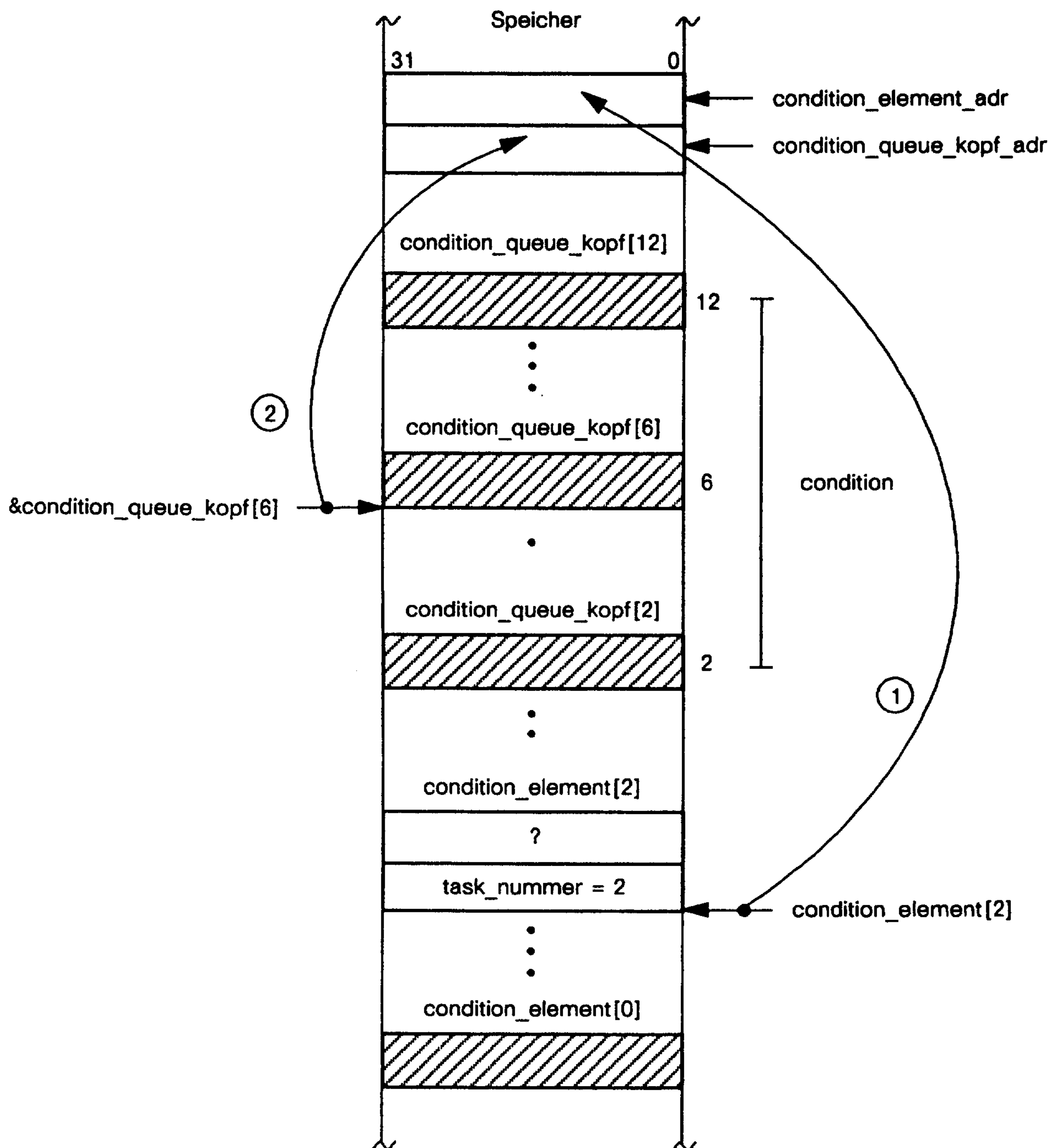


Bild 137: Adressen &condition_element[2] und &condition_queue_kopf[6] ermitteln

Als condition sei $cond_6$ = **warten_auf_s** mit der Identifikationsnummer **6** angenommen.
Führt z. B. Task_2 wait(warten_auf_s) aus, dann ergibt sich wegen aktuell_task_id = 2 als Adresse des korrespondierenden Condition-Elements &condition_element[2]. Diese Adresse wird in der Zeigervariablen condition_element_adr gespeichert, siehe (1).
Wegen warten_auf_s = 6 ergibt sich als Adresse des Condition_Queue_Kopf's &condition_queue_kopf[6]. Diese Adresse wird in der Zeigervariablen condition_queue_kopf_adr gespeichert, siehe (2).
Der Funktion **einketten** stehen jetzt alle Parameter zur Verfügung. Sie wird mit

- `einketten(condition_queue_kopf_adr,condition_element_adr);`

aufgerufen, und sie kettet das condition_element[2] an der Conditionstufe 6 in die Queue ein.
Danach ruft wait die Betriebssystem-Funktion suspendieren auf und übergibt ihr als Parameter die condition warten_auf_s. Sie kettet den Task-Control-Block aus der Task-Queue aus, bringt die Anwender-Task in den Zustand warten_auf_s und veranlaßt einen Taskwechsel.
Betrachten wir als nächstes die Betriebssystem-Funktion **signal**:

```
#define condition_max                    30
#include<struct3.h>

extern condition_queue_kopf_struct condition_queue_kopf[condition_max];
extern char aktuell_task_id,andere_task_id;

void signal(char condition)
{
 condition_element_struct    *condition_element_adr;

 if (condition_queue_kopf[condition].anzahl_der_condition_elemente != 0)
   {
    condition_element_adr
    = condition_queue_kopf[condition].erster_der_queue;
    andere_task_id = condition_element_adr->task_nummer;
    ausketten(&condition_queue_kopf[condition]);
    resume(andere_task_id,condition);
   }
}
```

Bild 138: Betriebssystem-Funktion **signal** (Datei condsig.c)

Wenn sie von einer Anwender-Task ausführt wird, dann stellt sie als erstes mit

- `if(condition_queue_kopf[condition].anzahl_der_condition_elemente != 0)`

fest, ob bei der spezifizierten condition (übergebener Parameter) Condition-Elemente eingekettet sind oder nicht. Sie überprüft zu diesem Zweck die **anzahl_der_condition_elemente** im condition_queue_kopf[condition]. Ist die Queue leer, bleibt signal passiv. Im anderen Fall ermittelt sie mit

- `condition_element_adr = condition_queue_kopf[condition].erster_der_queue;`

die Adresse des Condition-Elements am Anfang der Queue und speichert sie in der Zeigervariablen condition_element_adr.
Bild 139 zeigt hierfür ein Beispiel:

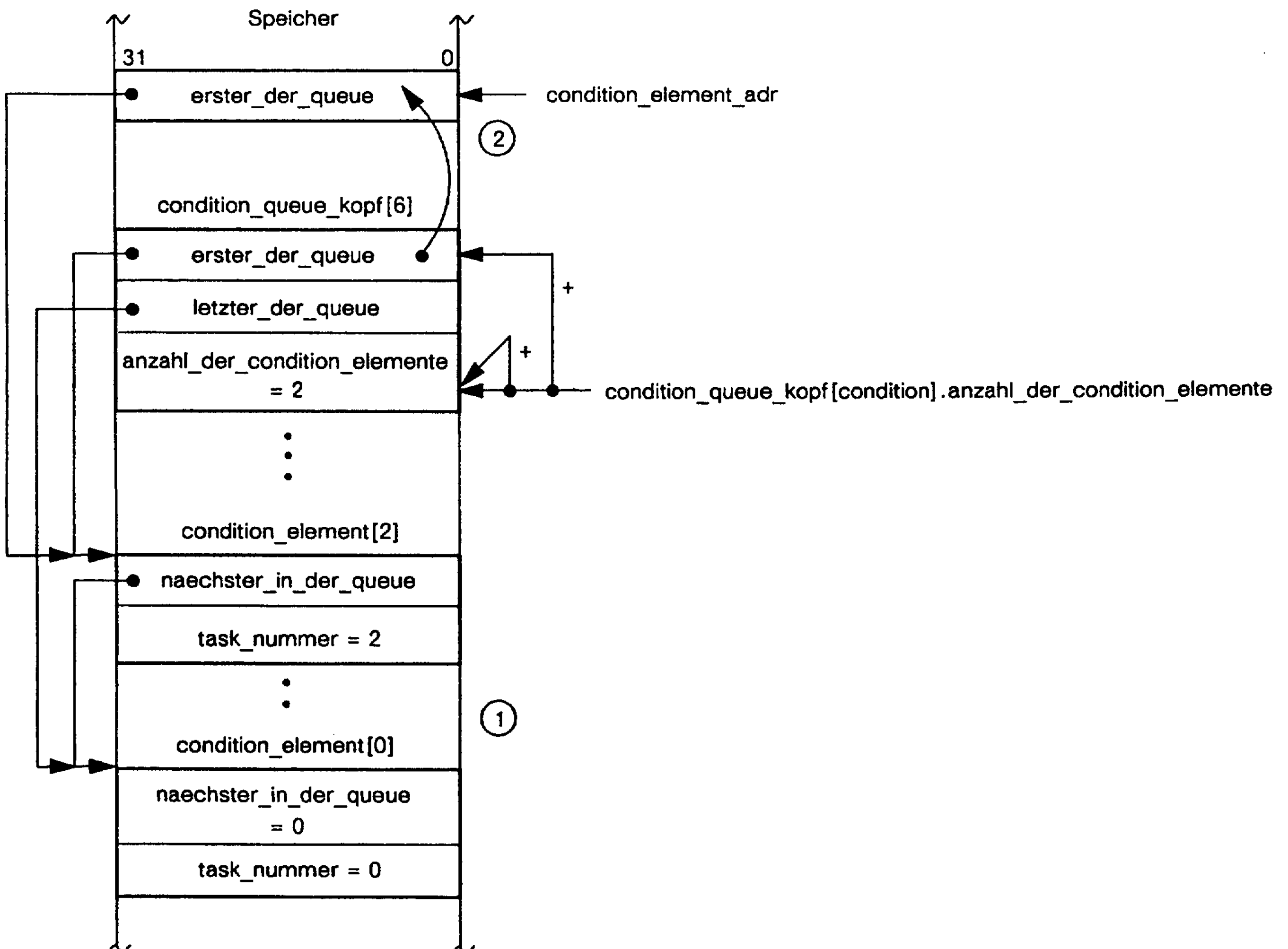


Bild 139: Adresse des ersten Condition-Elements ermitteln

Hier haben Task_2 und Task_0 nacheinander wait(warten_auf_s) ausgeführt, und so sind ihre Condition-Elemente nach dem FIFO-Prinzip an der Conditionstufe 6 eingekettet (1).
Beide Tasks befinden sich im Wartezustand und können entweder von Task_1 oder Task_3 aufgeweckt werden. Führt eine dieser Tasks signal(warten_auf_s) aus, dann speichert signal die Adresse des ersten Elements der Queue, also die Adresse von condition_element[2], in der Zeigervariablen condition_element_adr (2). Anschließend ermittelt signal mit

- `andere_task_id = condition_element_adr->task_nummer;`

die Nummer der aufzuweckenden Task und speichert sie in der externen char-Variablen **andere_task_id**, die in der Betriebssystem-Funktion **init** definiert ist (siehe Bild 133).
In unserem Beispiel ermittelt signal die Nummer der Task_2 (task_nummer=2):

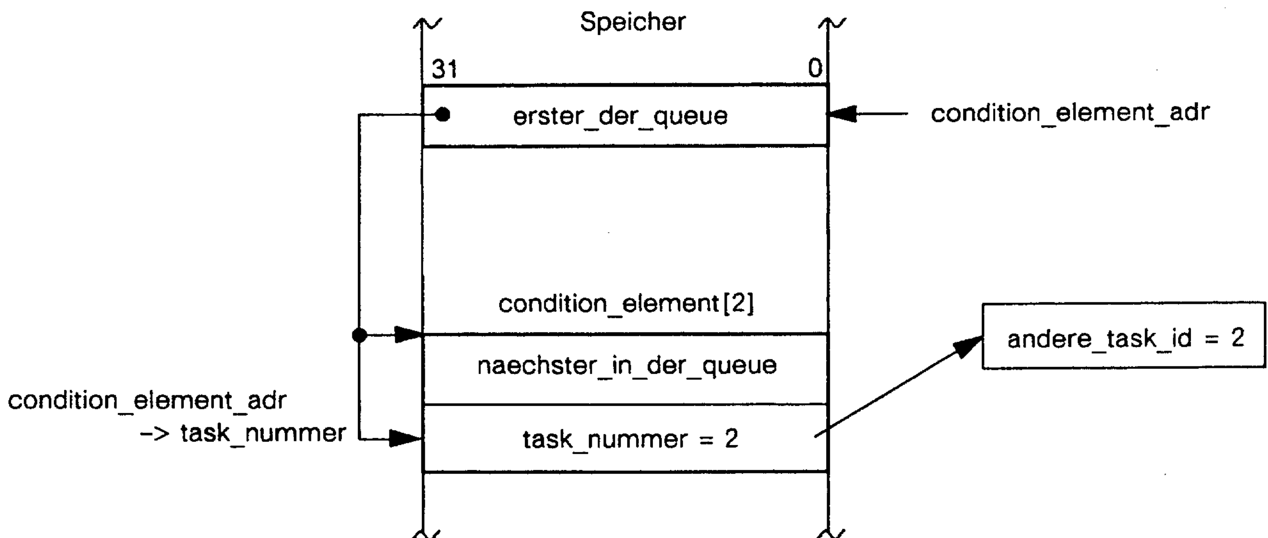


Bild 140: Nummer der aufzuweckenden Task ist 2

Nachdem dies geschehen ist, sorgt signal dafür, daß das Condition-Element der wartenden Task aus der Condition-Queue entfernt wird. Sie ruft deshalb die Betriebssystem-Funktion **ausketten** auf und übergibt ihr als Parameter die Adresse der Conditionstufe 6:

- `ausketten(&condition_queue_kopf[condition]);`

Danach aktiviert sie die Betriebssystem-Funktion **resume** und übergibt ihr als Parameter die Nummer der aufzuweckenden Task, nämlich andere_task_id=2 und die condition warten_auf_s:

- `resume(andere_task_id,condition);`

So bringt resume die Task_2 vom Zustand warten_auf_s in den Zustand ready, kettet ihren Task-Control-Block in die Task-Queue der Prioritätsstufe 2 ein (neu im System ankommende Task) und veranlaßt einen Taskwechsel.
Mit der Einführung von Condition-Queues kann jede V-Operationen ausführende Task irgendeine im Wartezustand befindliche Task ermitteln und aufwecken. So können sich zwei Tasks nicht mehr untereinander gegen die restlichen Tasks «verschwören» oder anders ausgedrückt, Aussperrungen können nicht mehr vorkommen. Damit die bisher für Task_0 und Task_1 «maßgeschneiderten» Betriebssystem-Funktionen **P_s** und **V_s** von **allen** Anwender-Tasks benutzt werden können, tauschen wir ihre P(s)/V(s)-Operationen gegen die P(sem)/V(sem)-Operationen aus.
Zunächst ersetzen wir in V(sem) die abstrakte Bedingung

```
if(beliebige_task == warten_auf_condition)
```

durch die allgemeine Bedingung

```
if(condition_queue_kopf[condition].anzahl_der_condition_elemente != 0)
```

und setzen außerdem **sem** = s und **condition** = warten_auf_s. So kommen wir schließlich zu folgenden Betriebssystem-Funktionen:

```
#define warten_auf_s 6

extern char s;

void P_s(void)
{
 if(s > 0)
    s --;
else
   wait(warten_auf_s);
}
```

Bild 141: Betriebssystem-Funktion P_s; verbesserte Version (Datei P_s.c)

```
#include<struct3.h>
#define condition_max  30
#define warten_auf_s    6

extern char s;
extern condition_queue_kopf_struct condition_queue_kopf[condition_max];

void V_s(void)
{
 if(condition_queue_kopf[warten_auf_s].anzahl_der_condition_elemente != 0)
    signal(warten_auf_s);
 else
     s ++;
}
```

Bild 142: Betriebssystem-Funktion V_s; verbesserte Version (Datei V_s.c)

Das von beiden Funktionen benutzte Semaphor s ist als externe char-Variable im Betriebssytem-Modul **init** definiert und dort mit **1** vorinitialisiert (siehe Bild 133).
P_s dekrementiert s bei s>0 und beläßt s bei s=0. So findet V_s für s **immer** eine 0 vor, die sie nur dann inkrementiert, wenn **keine** der anderen Tasks aus s wartet.
Die modifizierten Betriebssystem-Funktionen P_s und V_s wollen wir nun nutzen und den gegenseitigen Auschluß mehrerer Anwender-Tasks am Beispiel von Task_0..Task_3 realisieren. In den folgenden Bildern sind ihre Module zusammengestellt:

```
extern void far KN_write_io(short int,char *);
extern void far KN_P_s(void);
extern void far KN_V_s(void);
extern char far n;
static char b;
task_0_proc()
{
 while(1)
 {
  short int port_0=0x300;

  KN_P_s();                  /* P(s)  */
  b = n++;                   /* krit0 */
  KN_V_s();                  /* V(s)  */
  KN_write_io(port_0,&b);    /* rest0 */
 }
}
```

Bild 143: Anwender-Modul task_0 (Datei task_0.c)

```
extern void far KN_write_io(short int,char *);
extern void far KN_P_s(void);
extern void far KN_V_s(void);
extern char far n;
static char b;
task_1_proc()
{
while(1)
 {
  short int port_1=0x301;

  KN_P_s();                  /* P(s)  */
  b = n++;                   /* krit1 */
  KN_V_s();                  /* V(s)  */
  KN_write_io(port_1,&b);    /* rest1 */
 }
}
```

Bild 144: Anwender-Modul task_1 (Datei task_1.c)

```
extern void far KN_write_io(short int,char *);
extern void far KN_P_s(void);
extern void far KN_V_s(void);
extern char far n;
static char b;

task_2_proc()
{
 while(1)
 {
  short int port_2=0x302;

  KN_P_s();                  /* P(s)  */
  b = n++;                   /* krit2 */
  KN_V_s();                  /* V(s)  */
  KN_write_io(port_2,&b);    /* rest2 */
 }
}
```

Bild 145: Anwender-Modul task_2 (Datei task_2.c)

```
extern void far KN_write_io(short int,char *);
extern void far KN_P_s(void);
extern void far KN_V_s(void);
extern char far n;
static char b;

task_3_proc()
{
 while(1)
 {
  short int port_3=0x303;

  KN_P_s();                  /* P(s)  */
  b = n++;                   /* krit3 */
  KN_V_s();                  /* V(s)  */
  KN_write_io(port_3,&b);    /* rest3 */
 }
}
```

Bild 146: Anwender-Modul task_3 (Datei task_3.c)

Wie zu sehen ist, benutzen alle vier Anwender-Tasks die externe **far**-Variable n als gemeinsamen Zähler. Ihre kritischen Regionen krit0..krit3 schützen sie wie üblich durch die P_s/V_s-Funktionen, die sie über die Call-Gate-Deskriptoren KN_P_s(void) und KN_V_s(void) erreichen.

In den unkritischen Regionen rest0..rest3 bringen die Tasks den augenblicklichen Zählerstand über port_0..port_3 zur Anzeige. Um nun zu einem ablauffähigen System zu kommen, muß die bereits vorhandene Grundausstattung geringfügig modifiziert werden.

So muß als erstes BND386/486 von jeder Task ein **einzelnes** Objekt-Modul erzeugen. Da aber weder die Start-Task noch die Anwender-Tasks neue Module bekommen haben, ist dies nur für die OS_Kern_Task erforderlich.