Wichtigste BS-unterstützte Technik: das Semaphor (E.W.Dijkstra, 1965)

Wirkungsweise – hier: als Sperrvariable (spin lock) mit aktivem Warten:

Das allgemeine Semaphor:

```
void P(int *R)
{(*R)--; /*Res.-Anforderg!*/
  while (*R < 0);
}

void V(int *R)
{ (*R)++;
}
```

(Initialisierung mit `R=N;` für `N` Ressourcen)

Schutz kritischer Abschnitte, Kollisionsvermeidung

Das Binäre Semaphor (der Mutex):

```
void P(int *bS)
{ while (*bS==0);
  *bS=0;
}

void V(int *bS)
{ *bS=1;
}
```

(Initialisierung mit `bS=1;`)

Namen historisch gewachsen; bS äquivalent zu allgem. Semaphor für `N=1`

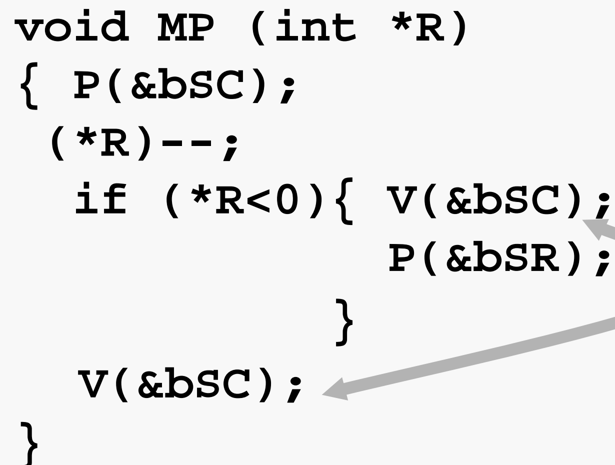
Anwendung:

```
P(&bS);   /* "Passieren"   */
/*...*/   /*krit.Abschnitt*/
V(&bS);   /* "Verlassen"   */
```

Konstruktion eines allgemeinen Semaphors aus zwei binären

[**R≤N** z.Z.verfügb.Ress.; **bSC** schützt R.-Zähler, **bSR** schützt R. selbst]:

(Allg.) Mehrfach-Semaphor:



```
void MP (int *R)
{ P(&bSC);
  (*R)--;
   if (*R<0){ V(&bSC);
              P(&bSR);
            }
   V(&bSC);
}
```

```
void MV (int *R)
{ P(&bSC);
  (*R)++;
   if (*R<=0) V(&bSR);
   else       V(&bSC);
}
```

Binäres Semaphor:

```
void P (int *bS)
{ while (*bS==0);
  *bS=0;
}
void V (int *bS)
{*bS=1;
}
```

Anwendung:

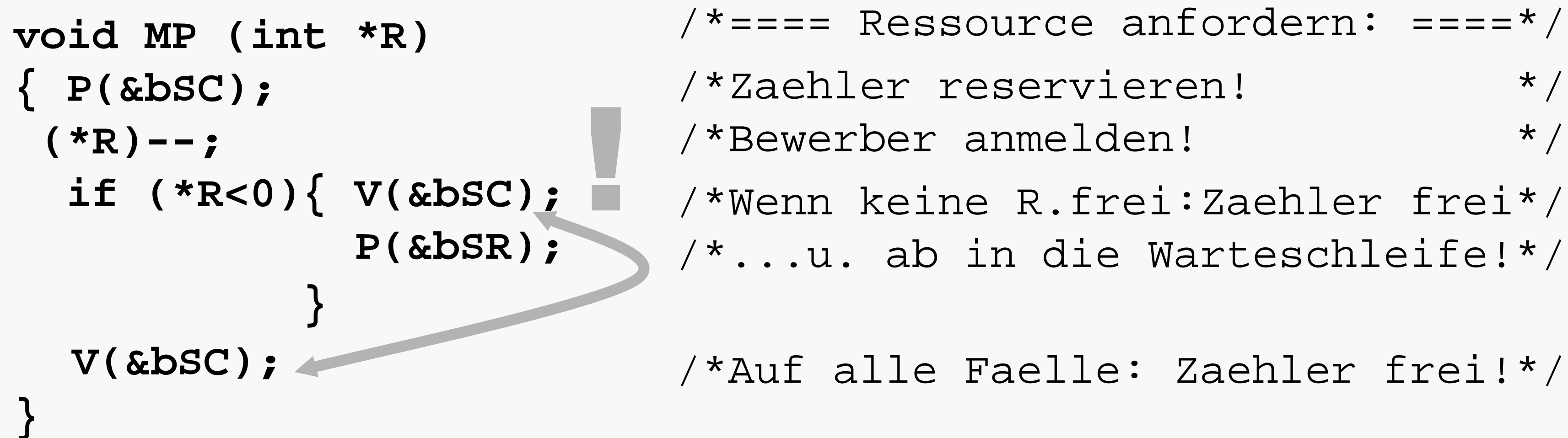
```
MP(&R);    /*  "Passieren"    */
/*...*/    /* krit.Abschnitt */
MV(&R);    /*  "Verlassen"    */
```

Initialisierungen: **R=N, bSC=1, bSR=0**

Konstruktion eines allgemeinen Semaphors aus zwei binären

[**R≤N** z.Z.verfügb.Ress.; **bSC** schützt R.-Zähler, **bSR** schützt R. selbst]:

(Allg.) Mehrfach-Semaphor:



```
void MP (int *R)                     /*==== Ressource anfordern: ====*/
{ P(&bSC);                           /*Zaehler reservieren!         */
  (*R)--;                            /*Bewerber anmelden!           */
   if (*R<0){ V(&bSC);               /*Wenn keine R.frei:Zaehler frei*/
              P(&bSR);               /*...u. ab in die Warteschleife!*/
            }
   V(&bSC);                          /*Auf alle Faelle: Zaehler frei!*/
}
```

```
void MV (int *R)                     /*==== Ressource verlassen: ====*/
{ P(&bSC);                           /*Zaehler reservieren          */
  (*R)++;                            /*Bewerber abmelden!           */
   if (*R<=0) V(&bSR);               /*Wenn nichts frei: Ress. frei! */
   else       V(&bSC);               /*Wenn was frei: Zaehler frei!  */
}
```

Bemerkungen zur Konstruktion eines allg. Semaphors aus zwei binären:

Ressourcen-Anforderung mit `MP()`:

- „Anmeldung" eines „Bewerbers" (Prozesses) erfolgt durch `R--;` Dazu wird der Zähler reserviert und immer in `MP()` wieder freigegeben.
- Bei `R>0` wird `bSR` gar nicht genutzt, sondern –nach Zähler-Freigabe– direkt auf die Ressource zugegriffen.
- Bei `R≤0` (wartender Prozeß) wird nach Freigabe des Zählers in die Warteschleife eingetreten; nach dem Austritt hieraus wird der Zähler freigegeben, der noch vom letzten Prozeß reserviert wurde.

Ressourcen-Freigabe mit `MV()`:

- „Abmeldung" erfolgt durch `R++;` Dazu wird der Zähler reserviert und nicht immer in `MV()` wieder freigegeben:
- Bei `R>0` wird `bSR` gar nicht genutzt, sondern nur der Zähler freigegeben.
- Bei `R≤0` (angemeldete/r wartende/r Prozess/e) wird die Ressource direkt freigegeben. Der Zähler bleibt reserviert und wird vom nächsten Prozeß (bei Zugriff auf die Ressource) freigegeben.

Bemerkungen zur Konstruktion eines allg. Semaphors aus zwei binären: (Fort)

Beobachtungen:

- Die `if`-Abfrage in `MP()` ist notwendig, damit sich der Prozeß nur dann über `P(bSR);` in eine Warteposition begibt (u. die Sperre `bSR=0` gesetzt wird), wenn die letzte Ressource bereits vergeben wurde (weshalb `R--;` einen negativen Wert ergeben hat).
- Zwischen Zähler-Reservierung in `MV()` durch den letzten und Zähler-Freigabe in `MP()` durch den nächsten Bewerber (falls `R≤0`) kann sich kein weiterer Prozeß anmelden („kleiner" Schutz vor Verhungern)
- Bei Absturz aller wartenden Prozesse zwischen Freigabe der Ressource durch den letzten Prozeß in `V(bSR)` und Freigabe des Zählers durch den nächsten kann sich kein weiterer Prozeß anmelden (Deadlock).
- Der vorgestellte Algorithmus löst elegant die Frage: „Wie läßt sich signalisieren, daß eine Ressource gerade frei wurde und ein wartender Prozeß darauf zugreifen kann, wenn insg. mehr Wartende als Ressourcen da sind (`R≤0`)?"

  Lösung: `bSR` wird in `MP()` und in `MV()` nur dann genutzt, wenn (und solange) alle verfügbaren Ressourcen vergeben sind (`if`-Abfragen mit `R<0` bzw. `R≤0`).

  Solange `R>0` ist, wird gar nicht „um Einlaß gebeten", sondern lediglich `R` dekrementiert, der Zähler freigegeben und die Ressource in Anspruch genommen. Entsprechend wird beim Verlassen `bSR=1` gesetzt, nur wenn `R≤0`.

Verlauf ab **R=1**, **bSC=1** und **bSR=0** für 3 Prozesse u. 1 Ressource:

```
void MP (int *R)
{ P(&bSC);
  (*R)--;
  if(*R<0){ V(&bSC);
            P(&bSR);
          }
   V(&bSC);
}
void MV (int *R)
{ P(&bSC);
  (*R)++;
   if(*R<=0)V(&bSR);
   else     V(&bSC);
}
void P(int *bS)
{ while (*bS==0);
  *bS=0;
}
void V(int *bS)
{*bS=1;
}
```

```
/*Initialisrg.*/
while (bSC==0);
bSC=0;
R--;
if(R<0)
{ bSC=1;
  while(bSR==0);
        bSR=0;
}
bSC=1;
/*krit.Abschn.*/
while (bSC==0);
bSC=0;
R++;
if(R<=0) bSR=1;
else     bSC=1;
```

Prozeß 1

R	bSC	bSR
1	1	0
1	0	0
0	0	0
0	1	0
-2 . .	1 . . .	0
-2	0	0
-1	0	0
-1	0	1

Time Slicing!

Prozeß 2

R	bSC	bSR
0	1	0
0	0	0
-1	0	0
-1	1	0
-1 . .	0 . . .	1
-1	0	0
-1	1	0
. . .	. . .	.
-1	0	0
0	0	0
0	0	1

Wunsch-denken!

Prozeß 3

R	bSC	bSR
-1	1	0
-1	0	0
-2	0	0
-2	1	0
0 . .	0 . . .	1
0	0	0
0	1	0
. . .	. . .	.
0	0	0
1	0	0
1	1	0

Bemerkungen zur vorausgegangenen Folie:

Die Anweisungen der zweiten Spalte sind wie Pseudocode verwendet (ohne Inhalts- o. Adreßoperatoren *, &), zum einen, um die Lesbarkeit des Beispiels zu verbessern, aber auch, um der Auflösung der Einzel-Funktionen Rechnung zu tragen.

Feststellungen:

- Versuchen mehrere Prozesse in den kritischen Bereich einzutreten, kann R aufgrund der Zähler-Freigabe V(&bSC) beliebig hohe negative Werte annehmen.
- Die Feststellung, daß eine Bedingung erfüllt ist und die Konsequenz o. Maßnahme dazu sind im Zeitscheibenverfahren keine atomare (d.h.: ununterbrechbare) Operation. Deshalb bieten manche Hersteller Makroinstruktionen wie „Test-and-Set", „Read-Modify-Write" oder „Test-and-Set-Lock" an. Alternativ dazu werden Interrupts gesperrt.
- Die o.a. Werte lassen sich auch im Debugger überprüfen, wenn zu den Funktionen der linken Spalte ein `main()` hinzugenommen wird – z.B.:

```
int main(void)
{ MP(&R); /*als 2. Prozess:*/ MP (&R); /*etc.*/
  printf ("Hallo, Welt!\n"); _getch();
  MV(&R);/*wird bei nur einem „Prozess" erreicht*/
}
```

Anmerkungen:

- Semaphore verhindern Kollisionen; sie regeln nicht, welcher unter mehreren wartenden Prozessen den Zugriff auf eine gerade freiwerdende Ressource erhält: Blockierung ist möglich!

- Semaphore können im Zeitscheiben-Verfahren ihre Wirkung verlieren, wenn zwischen Erfüllung einer Bedingung und der darauf folgenden Maßnahme andere Prozesse vom Scheduler aktiviert und Daten verändert werden.

- ANSI-C bietet für einige festgelegte Signale (innerhalb eines Prozesses!) d. Funktionen **`signal()`** u. **`raise()`** (**`signal.h`**):

```
void(*signal(int sig,void(*func)(int sig)))(int sig);
```

  liefert den Zeiger **`func`** auf die Funktion, die mit dem Argument **`sig`** ausgeführt wird, sobald das Signal **`sig`** ausgelöst wird; dies kann auch aus dem Programm geschehen durch:

```
int raise (int sig);
```

Zur Erinnerung (s.o.):

```
void (*func)(int);
```

bedeutet: **Zeigervariable** `func` bekommt als Werte Adressen v. Funktionen von `int` ohne Rückgabewert.

```
void (*func(...))(int);
```

bedeutet: **Funktion** namens `func(...)` hat als Rückgabewert Zeiger auf eine (zur Compilierzeit unbekannte) Funktion von `int` ohne Rückgabewert. (Funktionsadresse wird dem Zeiger erst zur Laufzeit zugewiesen.)

Entsprechend – „Lesehilfe" für den Prototyp von `signal()`:

```
void (*func) (int sig)
```

ist Zeiger auf beliebige Applikationsfunktion von `int`-Variable namens `sig`.

```
signal (int sig, void (*func) (int sig))
```

ist Funktion mit `int sig` und o.a. Funktionszeiger `func` als Parameter.

```
void(*signal(int sig,void(*func)(int sig)))(int sig);
```

bedeutet: Rückgabewert von `signal()` ist Zeiger auf Funktion ohne Rückgabewert mit `int sig` als Parameter (hier: Zeiger `func` selbst).

(s.a. `ProcSync\`)

- Beispiel:

```
#include <signal.h>
void atomic (int sig)
{ /* Quasi-atomare Funktion */
}
int main()
{ char happened=1;
   /*...*/
   /*atomic uebernimmt Ereignis CTRL+C:*/
   signal(SIGINT, atomic);        //Anmeldung atomic()
 (*signal(SIGINT, atomic))(SIGINT);//Anmeldung+Aufruf
  /*...*/
  /*CTRL+C ausloesen -> atomic-Aufruf:*/
  if (happened) raise (SIGINT);
  /*...*/
}
```

- **`signal`** und **`raise`** sind abhängig von Compiler und Implementierung

(s.a. `Atomic\`)