Anti-Deadlock-Maßnahmen:

Verhinderung zyklischer Wartebeziehung:

- Zentrale Warteschlange für Ressourcen-Zuweisung
- Zufallsabhängige Behandlung beteiligter Prozesse
- Einteilung von Prozeß-Gruppen zur Symmetrie-Brechung
- Zuteilung ungleicher Prioritäten

Verhinderung inkrementeller Akquisition:

- Gruppenweise Ressourcen-Reservierung
- Schutz kritischer Abschnitte

Prozeß-Interaktion u. -Kommunikation

Drei Forderungen an die Behandlung kritischer Abschnitte:

- Wechselseitiger Ausschluß (*mutual exclusion*):

  Zu keinem Zeitpunkt darf sich mehr als ein Prozeß (Thread) in seinem kritischen Abschnitt aufhalten.

- Fortschreiten (*progress*):

  Der Prozeß, der als nächster in seinen krit. Abschnitt eintreten darf, soll unter den bereits wartenden ausgewählt werden.

- Begrenztes Warten (*bounded waiting*):

  Einem wartenden Prozeß darf nur begrenzt viele Male der Eintritt in seinen kritischen Abschnitt verwehrt werden.

Lösungsansätze:

- Software-Lösungen (algorithmisch, z.B. innerhalb d. Sw-Umgebung)
- Hardware-Lösungen (Anwendung spezieller Hardware-Funktionen)
- Betriebssystem-unterstützte Lösungen (spezielle BS-Funktionen)

s.a. BS-Vorlesung

Prozeß-Interaktion u. -Kommunikation

Anmerkungen:

- **Software-Lösungen…**
  - … implizieren aktives Warten und daher Effizienz-Probleme

  ⇨ ... sind bei Ermangelung effizienterer Lösungen einzusetzen
- **Hardware-Lösungen…**
  - … als Interrupt-Sperrung bei Ein-Prozessor-Systemen
    - … vereiteln auch den Wechsel zu anderen Tasks (Multitasking)
    - … sind fatal bei Programmabstürzen u. Mißbrauch durch Programmierer
  - … als Interrupt-Sperrung bei Mehrprozessor-Systemen
    - … sind praktisch wirkungslos, wenn auf nur einen Prozessor beschränkt
  - … mit speziellen Hardware-Funktionen für Ein-Prozessor-Systeme
    - … ermöglichen wechselstg. Ausschl. mit atomaren read-modify-write-Fkt.
    - … verursachen bei anderen Prozessen aktives Warten (Effizienz-Probl.)
    - … lassen die Maßnahmen für Fortschreiten & begrenztes Warten offen
  - … mit speziellen Hardware-Funktionen für Mehrprozessor-Systeme
    - … können zudem wg. Alleinverfügung über d. Bus d. System blockieren

  ⇨ ... sind nützlich vor allem innerhalb des Betriebssystems

Wichtigste BS-unterstützte Lösung: 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)

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

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

  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. $R \leq 0$).

  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 $R \leq 0$.

Prozeß-Interaktion u. -Kommunikation

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;
```

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

Prozeß 1

Time Slicing !

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

Prozeß 2

Wunsch-denken !

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

Prozeß 3

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:

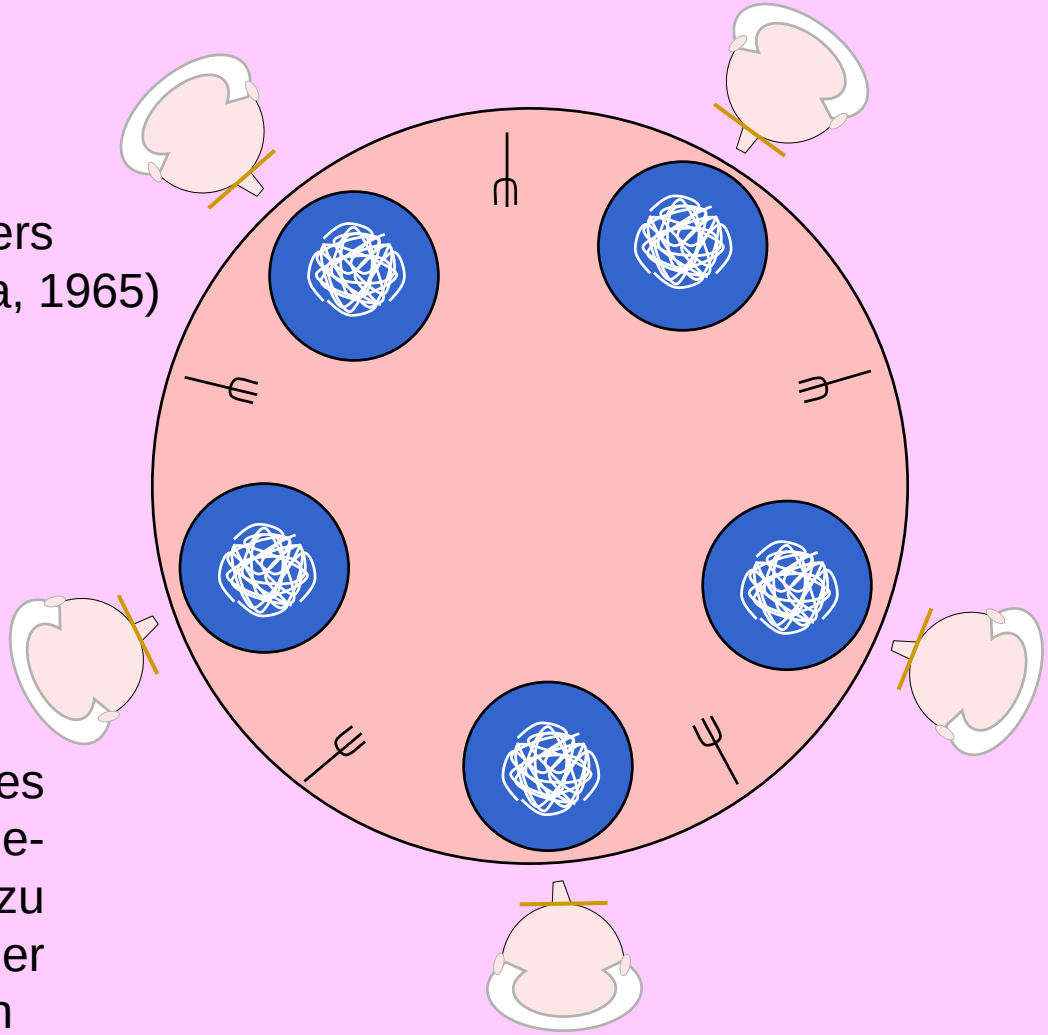
```
#include <signal.h>
void atomic (int sig)
{ /* Quasi-atomare Funktion */
}

int main()
{ char happened=1;
  /*...*/
  /*atomic uebernimmt Ereignis CTRL+C:*/
  signal (SIGINT, atomic);
  /*...*/
  /*CTRL+C ausloesen -> atomic-Aufruf:*/
  if (happened) raise (SIGINT);
  /*...*/
}
```

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

(s.a. Atomic\)

Das Problem der dinierenden Philosophen

(The Dining Philosophers Problem – E.W.Dijkstra, 1965)



Inzwischen klassisches Beispiel für alle Blockierungsarten und nahezu alle Fragestellungen der Prozeßsynchronisation

Geg.: 5 Philosophen sitzen an einem Tisch mit 5 Gabeln und 5 (Spaghetti-) Tellern; sie denken nach und essen abwechselnd; dabei braucht jeder 2 Gabeln.

Ges.: Algorithmus, der allen Philosophen regelmäßiges Essen sichert.

Hinweis:

- Ergreifen alle Philosophen gleichzeitig mit der gleichen Hand (li. o. re.) eine Gabel und warten, bis sie auch die andere bekommen, provozieren sie einen **Deadlock**.
- Erkennen sie daraufhin ihren Fehler (aber nicht den Deadlock) und wechseln wiederholt die Gabeln, so handelt es sich um einen **Livelock**.
- Greifen einem Philosophen seine Tischnachbarn ständig vor, so **verhungert** er.

(s.a. `Sophos\1SophStarv`)

3 Hauptstrategien zur Verhinderung von Blockierungen

bei den dinierenden Philosophen:

- Exklusiv nutzbare Ressourcen
- Nichtentziehbarkeit
- Inkrementelle Akquisition
- Zyklische Wartebeziehung

- **Zentralisierung** (*Centralization*)
  Steuernde Instanz - meist zusätzlicher Prozeß mit Pufferung
  (z.B.: Zentrale Warteschlange: nur ein Phil. darf jeweils essen)
- **Aufbrechen der Symmetrie** (*Breaking the symmetry*)
  Zuweisungsfolgen, die keine zyklischen Muster erzeugen
  (z.B.: Geradzahlige Indizes greifen zur li., andere zur re. Gabel)
- **Zufallsbezug** (*Randomization*)
  Zuweisungsfolgen abhängig vom Zufallsgenerator
  (z.B.: Losen, wer essen darf: „Wahrscheinlichkeit wird‘s richten“)

➡ Korrekte algorithmische Lösungen billiger und sicherer

Prozeß-Interaktion u. –Kommunikation: Übung:

Übung: Implementierung einer Lösung zum Problem der dinierenden Philosophen (s. Übungsblatt)

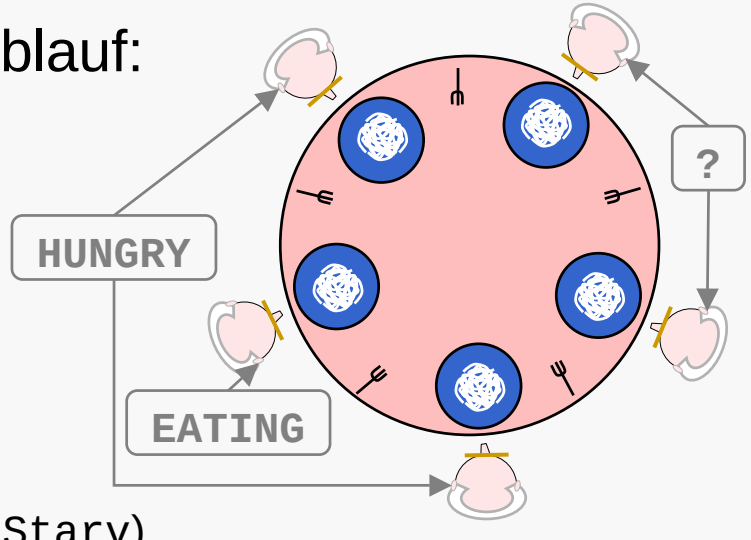
Ansatz: Einhaltung der Eß-Regel (Tischnachbarn essen nicht gleichzeitig) und der Forderung nach Fortschreiten (*progress* – hier: als nächster ißt einer der wartenden)

- ➡ Drei Zustände der Phil.: **`THINKING`**, **`HUNGRY`**, **`EATING`**

Beobachtung: (1)

Das Fortschreiten verlangsamt den Ablauf:

Während ein Philosoph ißt, können sich seine beiden Nachbarn hungrig melden; in diesem Fall können sich die verbleibenden zwei Philosophen nicht mehr hungrig melden, obwohl einer von ihnen essen könnte.



(s.a. `Sophos\2SophNoStarv`)

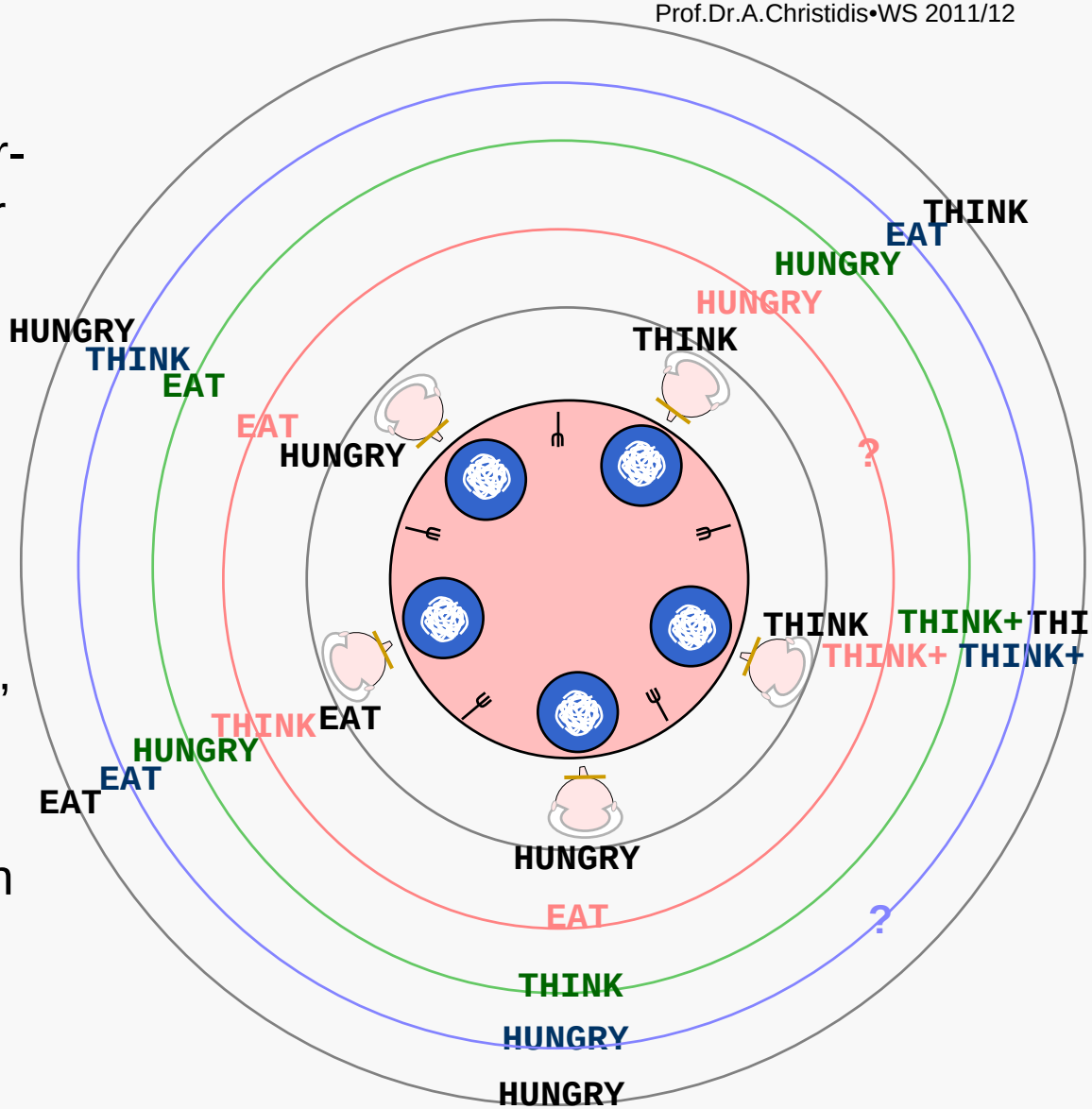
Prozeß-Interaktion u. –Kommunikation / Übung:

Beobachtung: (2)

- Ressourcen (Nachbar-Gabeln) individuell für jeden Philosophen
- ➡ keine „Standard-Semaphore“

Beobachtung: (3)

- max. 2 Philosophen **!** können jeweils essen,
- max. 2 können sich hungrig melden,
- mind. einer hat keinen Zustandswechsel („keinen Anspruch“)
- ➡ Verhungern möglich?

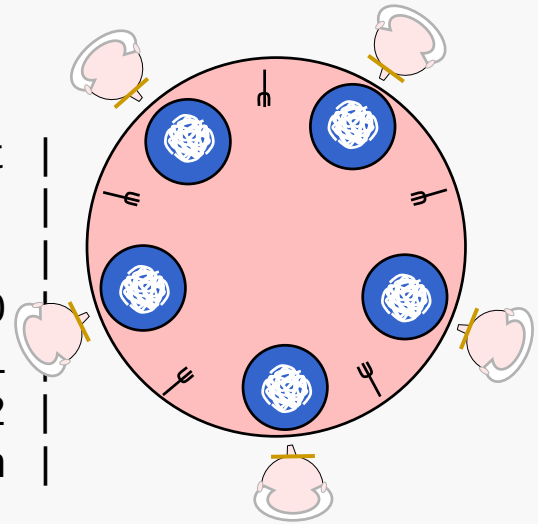


Implementierte Struktur:

```
typedef struct AllStat   /* (Strukturname)                          */
{ FILE *sema;            /* Semaphor-Datei                          */
  int   blockCS;         /* Blockierstatus krit. Abschnitt          */
  int   actId;           /* Index zuletzt blockierender Phil.       */
  int   philNr;          /* Anzahl dinierender Philosophen          */
  int   phHung[MAXPHIL]; /* Hungerstatus Philosoph [Index]          */
  int   closed;          /* Schliessung der Diner-Runde             */
  int   philId;          /* Index Philosoph im lfd. Programm        */
} SemaStat;              /* (Typname)                               */
```

Inhalt von Sema.txt:

```
1; blockCS:   Entsperrg/Sperrung krit. Abschnitt |
1; actId:     zuletzt blockierender Philosoph    |
3; philNr:    Anzahl dinierender Philosophen     |
2; phHung[0]: Denk-/Hunger-/EssStatus Phil.Idx 0 |
0; phHung[1]: Denk-/Hunger-/EssStatus Phil.Idx 1 |
1; phHung[2]: Denk-/Hunger-/EssStatus Phil.Idx 2 |
1; closed:    Diner eroeffnen/schliessen/beenden |
```



- Implementierungshinweis:

  Programmeinstellungen mit Bit-Charakter (Ein / Aus) werden oft sehr effizient als Einzel-Bit-Werte eines ganzzahligen Datentyps (**`unsigned int`**, **`int`**, **`char`**,...) codiert – ein Beispiel:

```
#define IRGENDWO   0 /*Werte fuer uTell*/
#define HIER       1 /*Potenzen von 2  */
#define JETZT      2 /*     "           */
#define IRGENDWANN 0 /*entbehrlich     */
```

```
#include <conio.h>
#include <stdio.h>
```

```
void WannUndWo (char uTell)
{ if (uTell &  HIER) printf("Hier ");
  else               printf("Irgendwo ");
  if (uTell & JETZT) printf("u. jetzt!\n");
  else               printf("u. irgendwann!\n");
}
int main()
{ WannUndWo(HIER|JETZT);   /*Ausgabe:"Hier u. jetzt!"     */
  WannUndWo(HIER|IRGENDWO);/*Ausgabe:"Hier u. irgendwann!"*/
  _getch();
}
```