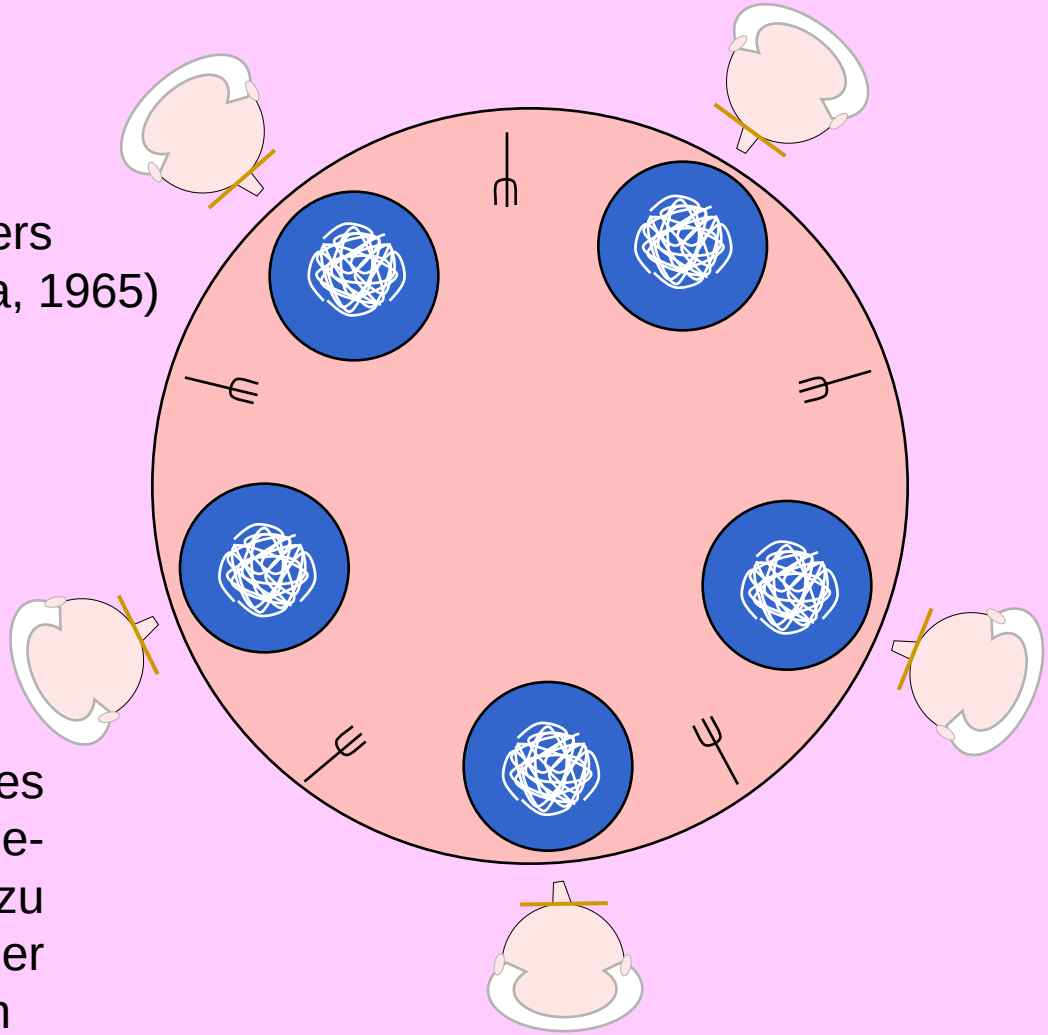
Das Problem der dinierenden Philosophen

(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

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

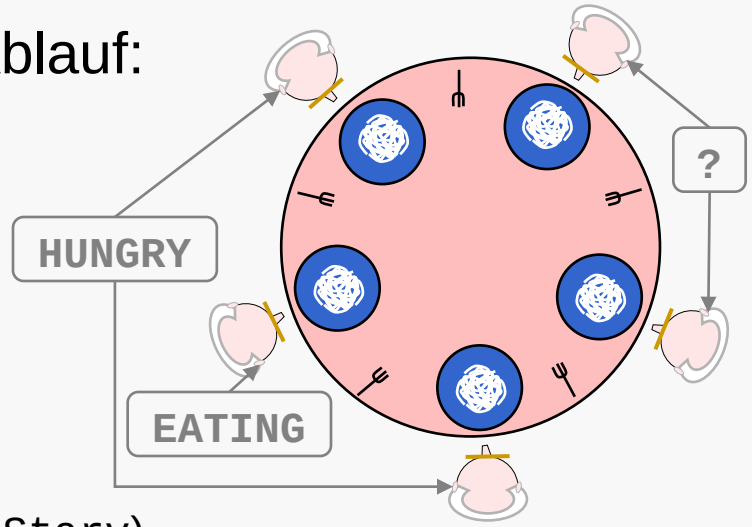
Ansatz: Einhaltung der Eß-Regel (Tischnachbarn essen nicht gleichzeitig) und der Forderung nach Fortschreiten (*progress* – hier: als nächster ißt einer der Wartenden)

➡ 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`)

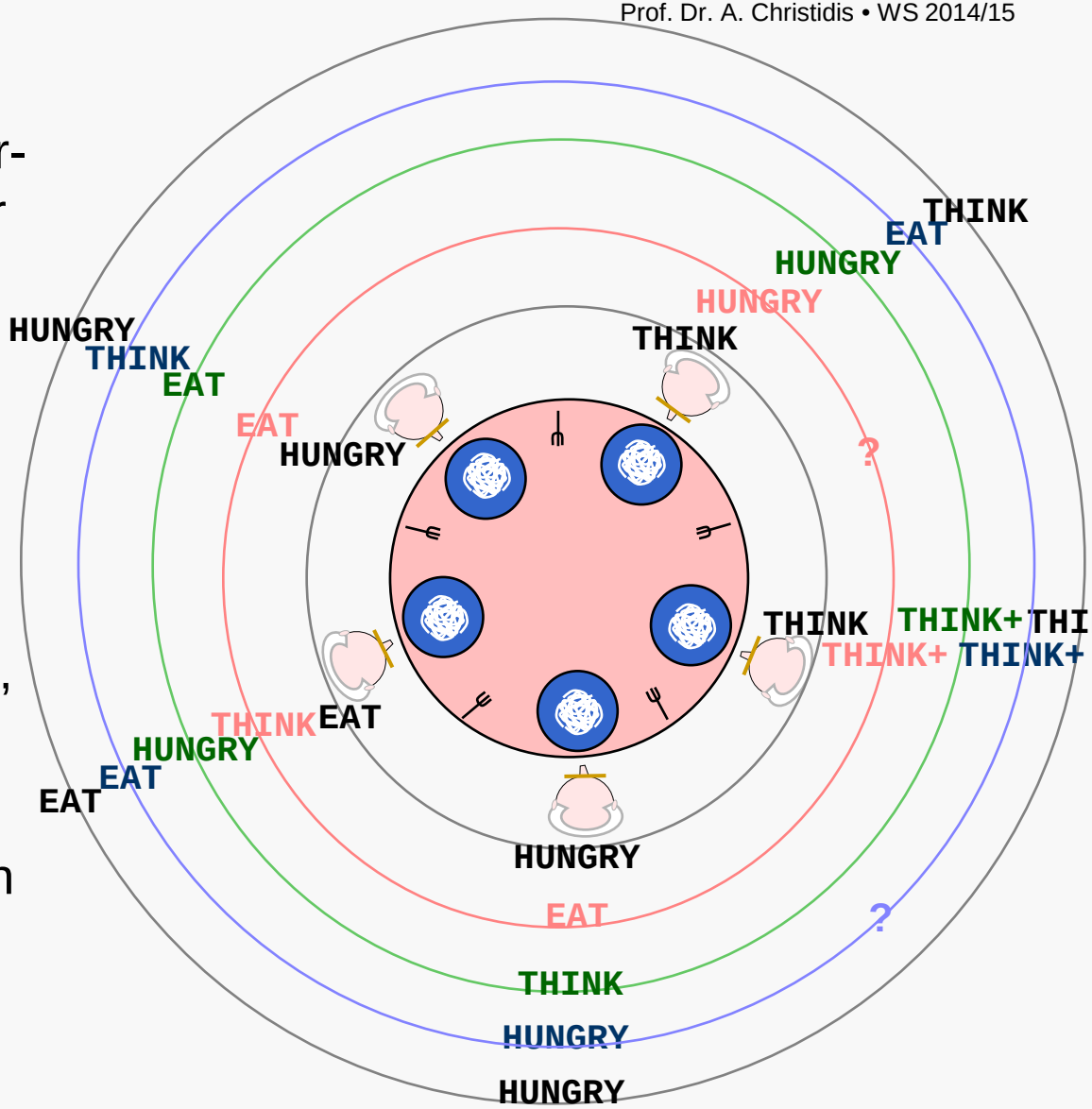
Interaktion u.-Kommunikation / Übung:

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?

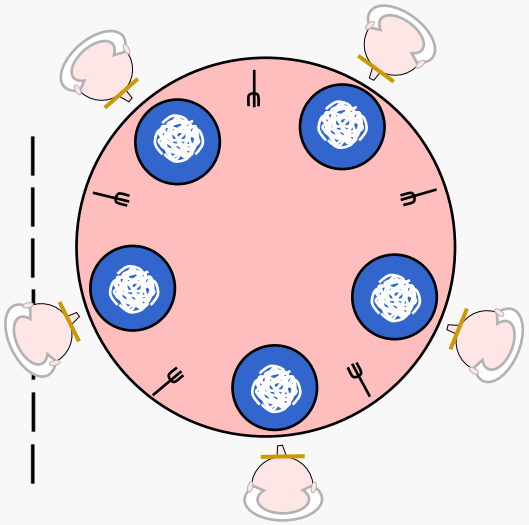


Implementierte Struktur:

```
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();
}
```