## 3.7 Klassische Koordinierungsprobleme

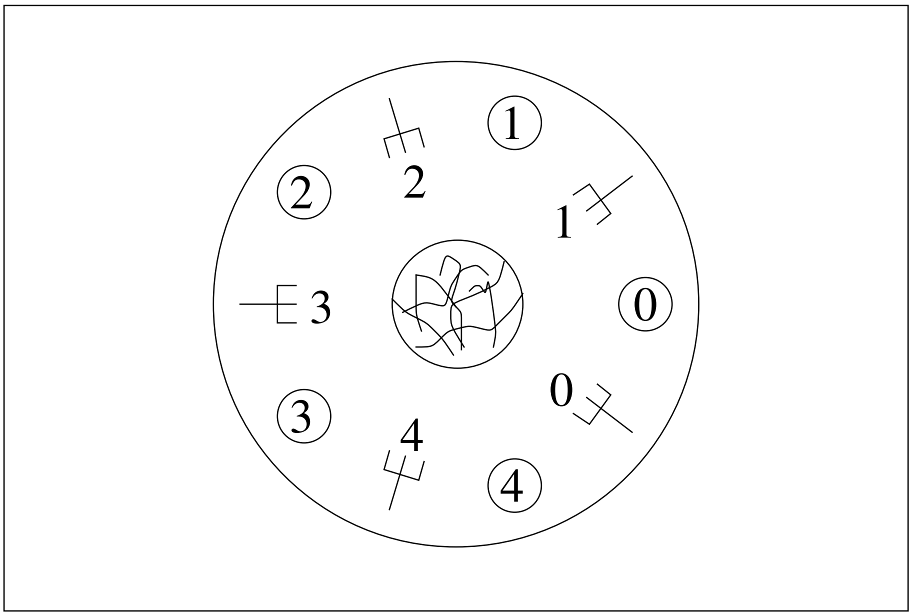
### 3.7.1 Das Problem der speisenden Philosophen (Dijkstra 1971)

- Die Aufgabenstellung:

  "The live of a philosopher consists of an alternation of thinking and eating:

```
CYCLE   BEGIN   think;
                eat
        END.
```

  Five philosophers, numbered from 0 through 4 are living in a house where the table is laid for them, each philosopher having his own place at the table:




  Their only problem - besides those of philosophy - is that the dish served is a very difficult kind of spaghetti, that has to be eaten with two forks. There are two forks next to each plate, so that presents no difficulty: as a consequence, however, no two neighbours may be eating simultaneously.

A very naive solution associates with each fork a binary semaphore with the initial value = 1 (indicating that the fork is free and, naming in each philosopher these semaphores in a local terminology, we could think the following solution for the philosopher's life adequate

```
CYCLE   BEGIN   think;
                P(left hand fork); P(right hand fork);
                    eat;
                V(left hand fork); V(right hand fork);
        END.
```

But this solution - although it guarantees that no two neighbours are eating simultaneously - must be rejected because it contains the danger of deadly embrace."

- Initialisierung

```
semaphore fork[5] = {semaphore(1), semaphore(1),
                     semaphore(1), semaphore(1),
                     semaphore(1)};
shared_int number_of_philosophers = 5;
```

- Philosoph

```
int thread_I()
{ int my_number = I;
  semaphore &left_fork = fork[I],
            &right_fork
              = fork[(I + 1)
                    % number_of_philosophers);

  while (TRUE) {
    cout << "Philosopher " << setw(1) << my_number
         << " is thinking      ----\n" << flush;
    sleep(1);
    cout << "Philosopher " << setw(1) << my_number
         << " is hungry        ++++\n" << flush;
    left_fork.P(); right_fork.P();
    cout << "Philosopher " << setw(1) << my_number
         << " is eating        ####\n" << flush;
    sleep(1);
    cout << "Philosopher " << setw(1) << my_number
         << " has enough       ----\n" << flush;
    left_fork.V(); right_fork.V();
  }
}
```

```
Philosopher 0 is thinking     ----
Philosopher 3 is thinking     ----
Philosopher 2 is thinking     ----
Philosopher 4 is thinking     ----
Philosopher 1 is thinking     ----
Philosopher 0 is hungry       ++++
Philosopher 3 is hungry       ++++
Philosopher 2 is hungry       ++++
Philosopher 4 is hungry       ++++
Philosopher 1 is hungry       ++++
```

**und es tut sich nichts mehr!**

- Konstruktion einer verklemmungsfreien Lösung
  - Koordinierungszustand

    denkend[i], hungrig[i], speisend[i] $0 \le i < 5$

    Änderung nur in kritischen Abschnitten
  - Verbotene Zustände V:

    $\exists i (0 \le i < 5 \wedge \text{speisend}[i] \wedge \mathit{speisend}[(i+1) \mathit{mod}\ 5])$
  - Systeminvariante:

    Eigenschaft des Systems, die immer dann erfüllt ist, wenn sich kein Aktivitätsträger (Prozeß) innerhalb eines kritischen Abschnittes aufhält.
  - Schwächste Systeminvariante: $\neg V$

    $$\neg(\exists i (0 \le i < 5 \wedge \mathit{speisend}[i] \wedge \mathit{speisend}[(i+1) \mathit{mod}\ 5]))$$

    $$=$$

    $$\forall i (0 \le i < 5 \Rightarrow \neg \mathit{speisend}[i] \vee \neg \mathit{speisend}[(i+1) \mathit{mod}\ 5])$$
  - Stabile und instabile Zustände

    Die Menge der erlaubten, d. h. nicht verbotenen, Zustände läßt sich unterteilen in

    - die Menge der instabilen Zustände, die zwar nicht als verboten zu betrachten sind, die jedoch möglichst schnell verlassen werden sollten, und
    - die Menge der stabilen Zustände, in denen das System beliebig lange verharren darf.

    $$\begin{aligned} \mathit{INST} = \exists i (0 \le i < 5 \wedge & \neg \mathit{speisend}[(i-1) \mathit{mod}\ 5] \\ & \wedge \mathit{hungrig[i]} \\ & \wedge \neg \mathit{speisend}[(i+1) \mathit{mod}\ 5]) \end{aligned}$$

    $$\mathit{STAB} = \neg V \wedge \neg \mathit{INST}$$

- Implementierungsstruktur:

Aktivitätsträger i

Wiederhole, solange das Endekriterium nicht erreicht ist

Koordinierungszustand neu einstellen
evtl. V(privater_sema[$j_1$])
V(privater_sema[$j_2$])
...

Der Eintritt in die nachfolgende Aktivität könnte einen verbotenen Zustand darstellen?

ja

nein

P(privater_sema[$j_i$])

Aktivität, die den Koordinierungszustand nicht ändert

- Initialisierung

```
#define THINKING 0
#define HUNGRY   1
#define EATING   2
shared_int number_of_philosophers = 5;
shared_int state[5] = {THINKING, THINKING,
                       THINKING, THINKING,
                       THINKING};
semaphore mutex(1);
semaphore private_sema[5]
            = {semaphore(0), semaphore(0),
               semaphore(0), semaphore(0),
               semaphore(0)};
```

- Philosoph

```
char state_text[3][2] = {"T", "H", "E"};
#define LEFT (i - 1 + 5) % 5
#define RIGHT (i + 1) % 5

void think(int i)
{ sleep(1);
}

void eat(int i)
{ int j;
  mutex.P();
  for (j = 0; j < 5; j++) {
    cout << state_text[state[j].value] << " ";
  }
  cout << "\n" << flush;
  mutex.V();
  sleep(1);
}
```

```
void test(int i)
{ if (state[i] == HUNGRY
      && state[(i + 4) % 5] != EATING
      && state[(i + 1) % 5] != EATING) {
    state[i] = EATING; private_sema[i].V();
  }
}

void take_forks(int i)
{ mutex.P();
    state[i] = HUNGRY;
    test(i);
  mutex.V();
  private_sema[i]->P();
}

void put_forks(int i)
{ mutex.P();
    state[i] = THINKING;
    test(LEFT); test(RIGHT);
  mutex.V();
}




thread_I()
{ int my_number = I;
  while (TRUE) {
    think(my_number);
    take_forks(my_number);
    eat(my_number);
    put_forks(my_number);
  }
}
```

```
T T T T E
T E T H E
H E H E T
E T H E T
E T E T H
T H E T E
T E T H E
H E T E T
E T H E T
E T E T H
T H E T E
T E T H E
H E T E T
E T H E T
E T E T H
T H E T E
T E T H E
T E T E T
E T H E T
E T E T H
T H E T E
T E T T E
H E T E T
E T H E T
E T E T H
T H E T E
```

## 3.7.2 Das Leser-Schreiber-Problem (Courtois e. a., 1971)

- Allgemeine Aufgabenstellung

  "We discuss two related problems of practical significance in which we recognize two classes of processes wishing to use the resource. The processes of the first class, named writers, must have exclusive access as in the original problem, but processes of the second class, the readers, may share the resource with an unlimited number of other readers."

- Problem 1

  "We demand of our solution that no reader be kept waiting unless a writer has already obtained permission to use the resource; i. e. no reader should wait simply because a writer ist waiting for other readers to finish."

- Initialisierung

```
shared_int readcount(0);
pv mutex(1), w(1);
```

- Leser

```
PROCESS_L()
{ while (1) {
      mutex.P();
        readcount++;
        if (readcount == 1) w.P();
      mutex.V();
        . . .   r e a d i n g
      mutex.P();
        readcount--;
        if (readcount == 0) w.V();
      mutex.V();
      remainder_L();
    }
}
```

- Schreiber

```
PROCESS_S()
{ while (1) {
      w.P();
        . . .   w r i t i n g
      w.V();
      remainder_W();
    }
}
```

- Problem 2

  "Here we retain the requirement that writers must have exclusive access while readers may share, but we add the requirement that once a writer is ready to write, he performs his write as soon as possible."

  - Initialisierung

```
shared_int readcount(0), writecount(0);
pv mutex1(1), mutex2(1), mutex3(1), w(1), r(1);
```

- Leser

```
PROCESS(0)
  shared_int readcount(0), writecount(4);
  pv mutex1(0), mutex2(1), mutex3(2), w(3), r(4);
  while (1) {
    mutex3.P();
      r.P();
        mutex1.P();
          readcount++; if (readcount == 1) w.P();
        mutex1.V();
      r.V();
    mutex3.V();
      . . .  r e a d i n g
    mutex1.P();
    readcount--; if (readcount == 0) w.V();
    mutex1.V();
  }
PROCESS_END
```

- Schreiber

```
PROCESS(1)
  shared_int readcount(0), writecount(4);
  pv mutex1(0), mutex2(1), mutex3(2), w(3), r(4);
  while (1) {
    mutex2.P();
    writecount++; if (writecount == 1) r.P();
    mutex2.V();
    w.P();
     . . .  w r i t i n g
    w.V();
    mutex2.P();
    writecount--; if (writecount == 0) r.V();
    mutex2.V();
    sleep(2);
  }
PROCESS_END
```