Parallele Programmiermodelle II

Stefan Lang

Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Universität Heidelberg
INF 368, Raum 532
D-69120 Heidelberg
phone: 06221/54-8264
email: Stefan.Lang@iwr.uni-heidelberg.de

WS 13/14

Parallele Programmiermodelle II

Kommunikation über gemeinsamen Speicher

- Barriere – Synchronization aller Prozeße
- Semaphore
- Philosphenproblem

Globale Synchronisation

- *Barriere*: Alle Prozessoren sollen aufeinander warten bis alle angekommen sind
- Barrieren werden häufig wiederholt ausgeführt:

```
while (1) {
    eine Berechnung;
    Barriere;
}
```

- Da die Berechnung lastverteilt ist, kommen alle gleichzeitig an der Barriere an
- Erste Idee: Zähle alle ankommenden Prozesse

Globale Synchronisation

Programm (Erster Vorschlag einer Barriere)

```
parallel barrier-1
{
    const int P=8;      int count=0;      int release=0;

    process Π [int p ∈ {0, ..., P − 1}]
    {
        while (1)
        {
            Berechnung;
            CSenter;                          // Eintritt
            if (count==0) release=0;          // Zurücksetzen
            count=count+1;                    // Zähler erhöhen
            CSexit;                           // Verlassen
            if (count==P) {
                count=0;                      // letzter löscht
                release=1;                    // und gibt frei
            }
            else while (release==0) ;         // warten
        }
    }
}
```

Barriere mit Richtungsumkehr

Warte *abwechselnd* auf *release*==1 und *release*==0

Programm (Barriere mit Richtungsumkehr)

```
parallel sense-reversing-barrier
{
    const int P=8;      int count=0;      int release=0;

    process Π [int p ∈ {0, ..., P − 1}]
    {
        int local_sense = release;
        while (1)
        {
            Berechnung;
            local_sense = 1-local_sense;        // Richtung wechseln
            CSenter;                            // Eintritt
            count=count+1;                      // Zähler erhöhen
            CSexit;                             // Verlassen
            if (count==P) {
                count=0;                        // letzter löscht
                release=local_sense;            // und gibt frei
            } else
                while (release≠local_sense) ;
        }
    }
}
```

Aufwand ist $O(P^2)$ da alle P Prozesse gleichzeitig durch einen kritischen Abschnitt müssen. Geht es besser?

Hierarchische Barriere: Variante 1

Bei der Barrier mit Zähler müssen alle P Prozesse durch einen kritischen Abschnitt. Dies erfordert $O(P^2)$ Speicherzugriffe. Wir entwickeln nun eine Lösung mit $O(P \log P)$ Zugriffen.

Wir beginnen mit *zwei* Prozessen und betrachten folgendes Programmsegment:

int *arrived*=0, *continue*=0;

Π_0:	Π_1:
	arrived=1;
while (¬*arrived*) ;	
arrived=0;	
continue=1;	
	while (¬*continue*) ;
	continue=0;

Wir verwenden zwei Synchronisationsvariablen, sogenannte *Flaggen*

Hierarchische Barriere: Variante 1

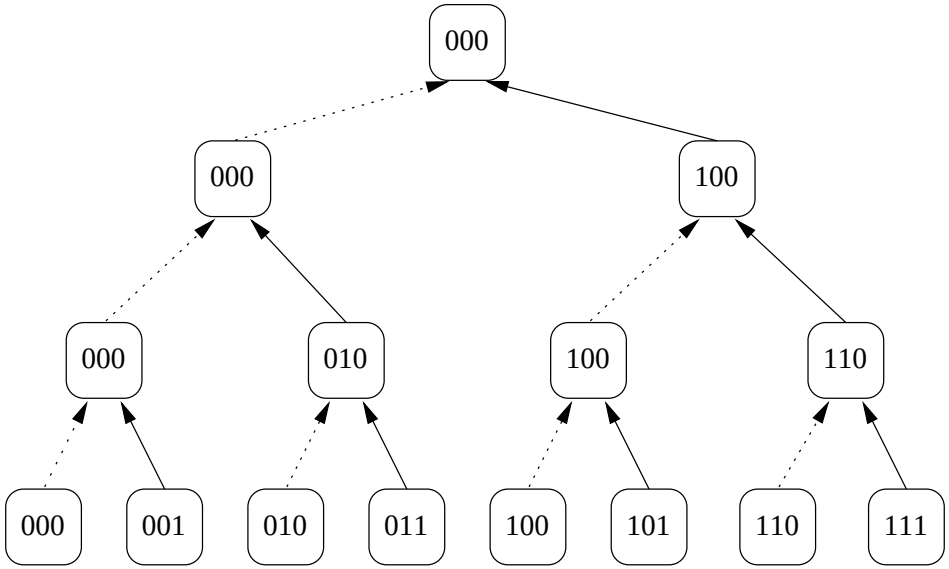
Bei Verwendung von Flaggen sind folgende Regeln zu beachten:

1. Der Prozess, der auf eine Flagge wartet setzt sie auch zurück.
2. Eine Flagge darf erst erneut gesetzt werden, wenn sie sicher zurückgesetzt worden ist.

Beide Regeln werden von unserer Lösung beachtet

Die Lösung nimmt sequentielle Konsistenz des Speichers an!

Wir wenden diese Idee nun hierarchisch an:



Hierarchische Barriere: Variante 1

Programm (Barriere mit Baum)

parallel *tree-barrier*
{

const int $d = 4$, $P = 2^d$; **int** *arrived[P]*=$\{0[P]\}$, *continue[P]*=$\{0[P]\}$;

process Π [**int** $p \in \{0, ..., P-1\}$]
{

int *i, r, m, k;*
while *(1) {*
Berechnung;
for $(i = 0; i < d; i{+}{+})$ *{* // aufwärts
$r = p \,\&\, \left[\sim \left(\sum_{k=0}^{i} 2^k\right)\right]$; // Bits 0 bis i löschen
$m = r \mid 2^i$; // Bit i setzen
if $(p == m)$ *arrived[m]=1;*
if $(p == r)$ *{*
while$(\neg$*arrived[m]) ;* // warte
arrived[m]=0;
}
} *// Prozess 0 weiss, dass alle da sind*
...

Hierarchische Barriere: Variante 1

Programm (Barriere mit Baum cont.)

parallel *tree-barrier cont.*
{

```
        ...
            for (i = d − 1; i ≥ 0; i − −) {          // abwärts
                r = p & [∼ (Σ_{k=0}^{i} 2^k)];       // Bits 0 bis i löschen
                m = r | 2^i;
                if (p == m) {
                    while(¬continue[m]) ;
                    continue[m]=0;
                }
                if (p == r) continue[m]=1;
            }
        }
    }
}
```

Achtung: Flaggenvariablen sollten in verschiedenen Cache-Lines sein, damit sich Zugriffe nicht behindern!

Hierarchische Barriere: Variante 2

Diese Variante stellt eine symmetrische Lösung der Barriere mit *rekursiver Verdopplung* vor.

Wir betrachten wieder zunächst die Barriere für zwei Prozesse Π_i und Π_j:

Π_i:	Π_j:
while (*arrived*[*i*]) ;	**while** (*arrived*[*j*]) ;
arrived[*i*]=1;	*arrived*[*j*]=1;
while (¬*arrived*[*j*]) ;	**while** (¬*arrived*[*i*]) ;
arrived[*j*]=0;	*arrived*[*i*]=0;

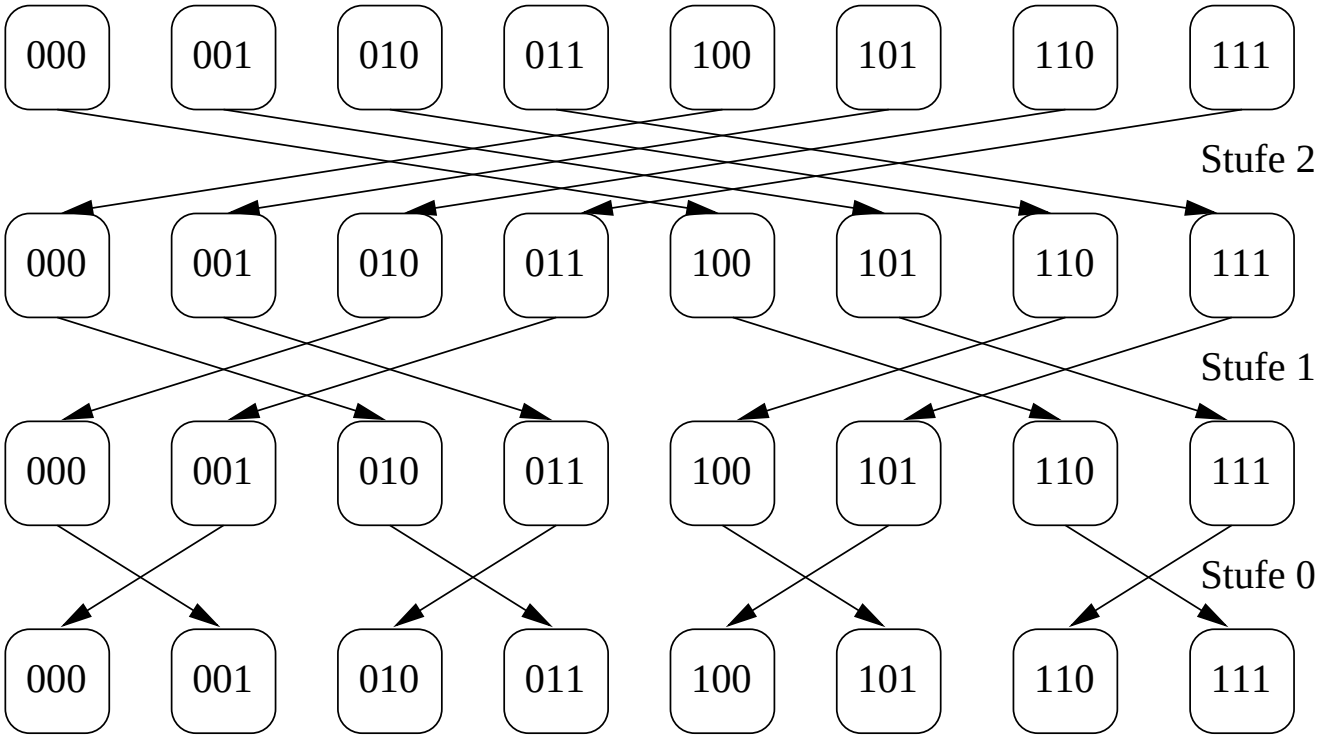
Im Vorgriff auf die allgemeine Lösung sind die Flaggen als Feld organisiert, zu Beginn sind alle Flaggen 0.

Ablauf in Worten:

- Zeile 2: Jeder setzt *seine* Flagge auf 1
- Zeile 3: Jeder wartet auf die Flagge des anderen
- Zeile 4: Jeder setzt die Flagge des *anderen* zurück
- Zeile 1: Wegen Regel 2 von oben warte bis Flagge zurückgesetzt ist
- Nun wenden wir die Idee rekursiv an!

Hierarchische Barriere: Variante 2

Rekursive Verdopplung verwendet folgende Kommunikationsstruktur:



- Keine untätigen Prozessoren
- Jeder Schritt ist eine Zweiwegkommunikation

Hierarchische Barriere: Variante 2

Programm (Barriere mit rekursiver Verdopplung)

```
parallel recursive-doubling-barrier
{
    const int d = 4,      P = 2^d; int arrived[d][P]={0[P · d]};

    process Π [int p ∈ {0, ..., P − 1}]
    {
        int i, q;
        while (1) {
            Berechnung;
            for (i = 0; i < d; i++)                    // alle Stufen
            {
                q = p ⊕ 2^i;                           // Bit i umschalten
                while (arrived[i][p]) ;
                arrived[i][p]=1;
                while (¬arrived[i][q]) ;
                arrived[i][q]=0;
            }
        }
    }
}
```

Semaphore

Eine Semaphore ist eine Abstraktion einer Synchronisationsvariable, die die elegante Lösung einer Vielzahl von Synchronisationsproblemen erlaubt

Alle bisherigen Programme haben *aktives Warten* verwendet. Dies ist sehr ineffizient bei quasi-paralleler Abarbeitung mehrerer Prozesse auf einem Prozessor (multitasking). Die Semaphore erlaubt es Prozesse in den Wartezustand zu versetzen.

Wir verstehen eine Semaphore als abstrakten Datentyp: Datenstruktur mit Operationen, die gewisse Eigenschaften erfüllen:

Eine Semaphore S hat einen ganzzahligen, nichtnegativen Wert $value(S)$, der beim Anlegen der Semaphore mit dem Wert *init* belegt wird.

Auf einer Semaphore S sind zwei Operationen **P**(S) und **V**(S) definiert mit:

- **P**(S) erniedrigt den Wert von S um eins falls $value(S) > 0$, sonst *blockiert* der Prozess solange bis ein anderer Prozess eine **V**–Operation auf S ausführt.
- **V**(S) befreit einen anderen Prozess aus seiner **P**–Operation falls einer wartet (warten mehrere wird einer ausgewählt), ansonsten wird der Wert von S um eins erhöht. **V**–Operationen blockieren nie!

Semaphore

Ist die Zahl *erfolgreich beendeter* **P**-Operationen n_P und die der **V**-Operationen n_V, so gilt für den Wert der Semaphore immer:

$$value(S) = n_V + init - n_P \geq 0$$

oder äquivalent $n_P \leq n_V + init$.

Der Wert einer Semaphore ist nach aussen *nicht* sichtbar. Er äußert sich nur durch die Ausführbarkeit der **P**–Operation

Das Erhöhen bzw. Erniedrigen einer Semaphore erfolgt atomar, mehrere Prozesse können also **P**/**V**–Operationen gleichzeitig durchführen

Semaphore, die einen Wert größer als eins annehmen können bezeichnet man als *allgemeine Semaphore*

Semaphore, die nur Werte $\{0, 1\}$ annehmen, heißen *binäre Semaphore*

Notation:

Semaphore *S*=1;
Semaphore *forks*[5] = {1 [5]};

Wechselseitiger Ausschluss mit Semaphore

Wir zeigen nun wie alle bisher behandelten Synchronisationsprobleme mit Semaphorvariablen gelöst werden können und beginnen mit wechselseitigem Ausschluss unter Verwendung von einer einzigen binären Semaphore:

Programm (Wechselseitiger Ausschluss mit Semaphore)

```
parallel cs-semaphore
{
    const int P=8;
    Semaphore mutex=1;
    process Π [int i ∈ {0, ..., P − 1}]
    {
        while (1)
        {
            P(mutex);
            kritischer Abschnitt;
            V(mutex);
            unkritischer Abschnitt;
        }
    }
}
```

Wechselseitiger Ausschluss mit Semaphore

Bei Multitasking können die Prozesse in den Zustand wartend versetzt werden

Fairness ist leicht in den Aufweckmechanismus zu integrieren (FCFS)

Speicherkonsistenzmodell kann von der Implementierung beachtet werden, Programme bleiben portabel (z. B. Pthreads)

Barriere mit Semaphore

- Jeder Prozess muss verzögert werden bis der andere an der Barriere ankommt.
- Die Barriere muss wiederverwendbar sein, da sie in der Regel wiederholt ausgeführt wird.

Programm (Barriere mit Semaphore für zwei Prozesse)

```
parallel barrier-2-semaphore
{
    Semaphore b1=0, b2=0;
    process Π1
    {
        while (1) {
            Berechnung;
            V(b1);
            P(b2);
        }
    }

    process Π2
    {
        while (1) {
            Berechnung;
            V(b2);
            P(b1);
        }
    }
}
```

Barriere mit Semaphore

Rollen wir die Schleifen ab, dann sieht es so aus:

Π_1:

Berechnung 1;
V(*b1*);
P(*b2*);
Berechnung 2;
V(*b1*);
P(*b2*);
Berechnung 3;
V(*b1*);
P(*b2*);
...

Π_2:

Berechnung 1;
V(*b2*);
P(*b1*);
Berechnung 2;
V(*b2*);
P(*b1*);
Berechnung 3;
V(*b2*);
P(*b1*);
...

Angenommen Prozess Π_1 arbeitet an Berechnungsphase i, d.h. er hat **P**(*b2*) $i-1$-mal ausgeführt. Angenommen Π_2 *arbeitet an* Berechnungphase $j < i$, d.h. er hat **V**(*b2*) $j-1$ mal ausgeführt, somit gilt

$$n_P(b2) = i - 1 > j - 1 = n_V(b2).$$

Andererseits stellen die Semaphorenregeln sicher, dass

$$n_P(b2) \leq n_V(b2) + 0.$$

Dies ist ein Widerspruch und es kann nicht $j < i$ gelten. Das Argument ist symmetrisch und gilt auch bei Vertauschen der Prozessornummern.

Erzeuger/Verbraucher $m/n/1$

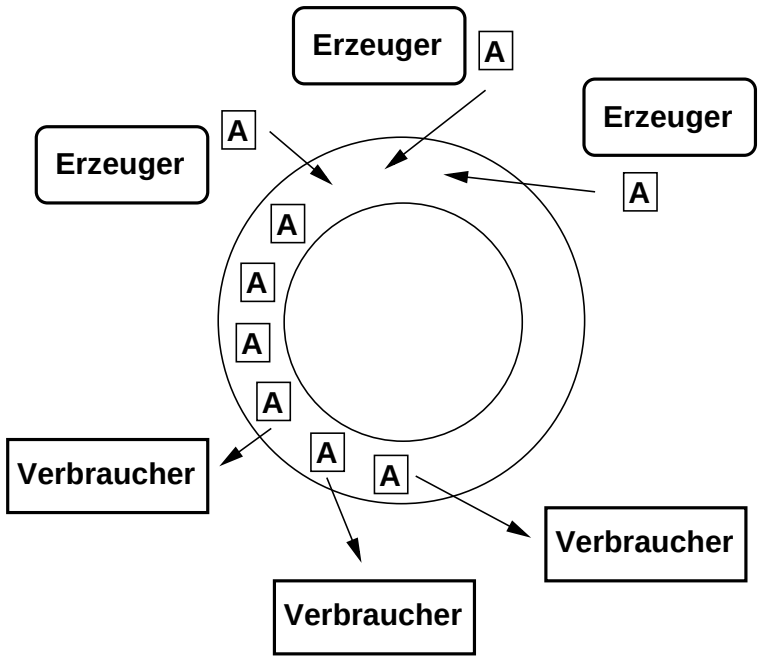
m Erzeuger, *n* Verbraucher, 1 Pufferplatz,

Erzeuger muss blockieren wenn Pufferplatz besetzt ist

Verbraucher muss blockieren wenn kein Auftrag da ist

Wir benutzen zwei Semaphore:

- *empty*: zählt Anzahl *freie* Pufferplätze
- *full*: zählt Anzahl *besetzte* Plätze (Aufträge)



Erzeuger/Verbraucher $m/n/1$

Programm (m Erzeuger, n Verbraucher, 1 Pufferplatz)

```
parallel prod-con-nm1
{
    const int m = 3, n = 5;
    Semaphore empty=1;                    // freier Pufferplatz
    Semaphore full=0;                     // abgelegter Auftrag
    T buf;                                // der Puffer
    process P [int i ∈ {0, ..., m − 1}] {
        while (1) {
            Erzeuge Auftrag t;
            P(empty);                     // Ist Puffer frei?
            buf = t;                      // speichere Auftrag
            V(full);                      // Auftrag abgelegt
        }
    }
    process C [int j ∈ {0, ..., n − 1}] {
        while (1) {
            P(full);                      // Ist Auftrag da?
            t = buf;                      // entferne Auftrag
            V(empty);                     // Puffer ist frei
            Bearbeite Auftrag t;
        }
    }
}
```

Geteilte binäre Semaphore (*split binary semaphore*):

$$0 \leq empty + full \leq 1 \quad \text{(Invariante)}$$

Erzeuger/Verbraucher $1/1/k$

1 Erzeuger, 1 Verbraucher, k Pufferplätze,

Puffer ist Feld der Länge k vom Typ T. Einfügen und Löschen geht mit

$$buf[front] = t; \quad front = (front + 1) \mod k;$$

$$t = buf[rear]; \quad rear = (rear + 1) \mod k;$$

Semaphore wie oben, nur mit k initialisiert!

Programm (1 Erzeuger, 1 Verbraucher, k Pufferplätze)

```
parallel prod-con-11k
{
    const int k = 20;
    Semaphore empty=k;          // zählt freie Pufferplätze
    Semaphore full=0;           // zählt abgelegte Aufträge
    T buf[k];                   // der Puffer
    int front=0;                // neuester Auftrag
    int rear=0;                 // ältester Auftrag

}
```

Erzeuger/Verbraucher $1/1/k$

Programm (1 Erzeuger, 1 Verbraucher, k Pufferplätze)

```
parallel prod-con-11k
{
    process P {
        while (1) {
            Erzeuge Auftrag t;
            P(empty);                    // Ist Puffer frei?
            buf[front] = t;              // speichere Auftrag
            front = (front+1) mod k;     // nächster freier Platz
            V(full);                     // Auftrag abgelegt
        }
    }
    process C {
        while (1) {
            P(full);                     // Ist Auftrag da?
            t = buf[rear];               // entferne Auftrag
            rear = (rear+1) mod k;       // nächster Auftrag
            V(empty);                    // Puffer ist frei
            Bearbeite Auftrag t;
        }
    }
}
```

Erzeuger/Verbraucher $m/n/k$

m Erzeuger, n Verbraucher, k Pufferplätze,

Wir müssen nur sicherstellen, dass Erzeuger untereinander und Verbraucher untereinander nicht gleichzeitig den Puffer manipulieren

Benutze zwei zusätzliche binäre Semaphore *mutexP* und *mutexC*

Programm (m Erzeuger, n Verbraucher, k Pufferplätze)

```
parallel prod-con-mnk
{
    const int k = 20, m = 3, n = 6;
    Semaphore empty=k;      // zählt freie Pufferplätze
    Semaphore full=0;       // zählt abgelegte Aufträge
    T buf[k];               // der Puffer
    int front=0;            // neuester Auftrag
    int rear=0;             // ältester Auftrag
    Semaphore mutexP=1;     // Zugriff der Erzeuger
    Semaphore mutexC=1;     // Zugriff der Verbraucher

}
```

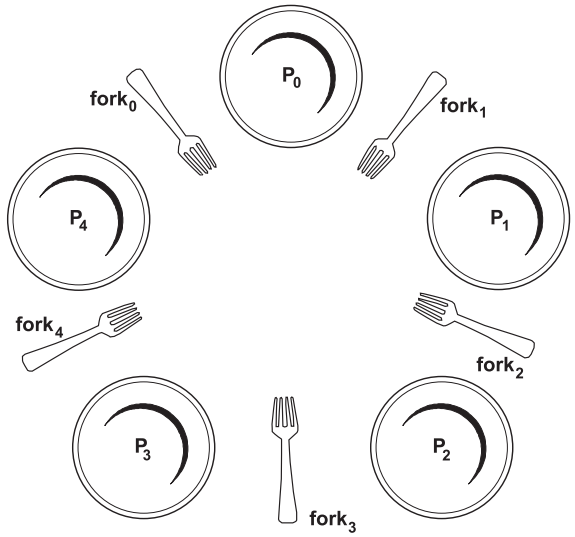
Erzeuger/Verbraucher $m/n/k$

Programm (m Erzeuger, n Verbraucher, k Pufferplätze)

```
parallel process
{
    P [int i ∈ {0, ..., m − 1}] {
        while (1) {
            Erzeuge Auftrag t;
            P(empty);                          // Ist Puffer frei?
            P(mutexP);                         // manipuliere Puffer
            buf[front] = t;                    // speichere Auftrag
            front = (front+1) mod k;           // nächster freier Platz
            V(mutexP);                         // fertig mit Puffer
            V(full);                           // Auftrag abgelegt
        }
    }
    process C [int j ∈ {0, ..., n − 1}] {
        while (1) {
            P(full);                           // Ist Auftrag da?
            P(mutexC);                         // manipuliere Puffer
            t = buf[rear];                     // entferne Auftrag
            rear = (rear+1) mod k;             // nächster Auftrag
            V(mutexC);                         // fertig mit Puffer
            V(empty);                          // Puffer ist frei
            Bearbeite Auftrag t;
        }
    }
}
```

Speisende Philosophen

Komplexere Synchronisationsaufgabe: Ein Prozess benötigt exklusiven Zugriff auf mehrere Resourcen um eine Aufgabe durchführen zu können.
$\rightarrow$ Überlappende kritische Abschnitte.



Fünf Philosophen sitzen an einem runden Tisch. Die Tätigkeit jedes Philosophen besteht aus den sich abwechselnden Phasen des Denkens und des Essens. Zwischen je zwei Philosophen liegt eine Gabel und in der Mitte steht ein Berg Spaghetti. Zum Essen benötigt ein Philosoph zwei Gabeln – die links und *rechts von ihm liegende.*

Speisende Philosophen

Das Problem:
Schreibe ein paralleles Programm, mit einem Prozess pro Philosoph, welches

- einer maximalen Zahl von Philosophen zu Essen erlaubt und
- das eine Verklemmung vermeidet

Grundgerüst eines Philosophen:

```
while (1)
{
    Denke;
    Nehme Gabeln;
    Esse;
    Lege Gabeln zurück;
}
```

Naive Philosophen

Programm (Naive Lösung des Philosophenproblems)

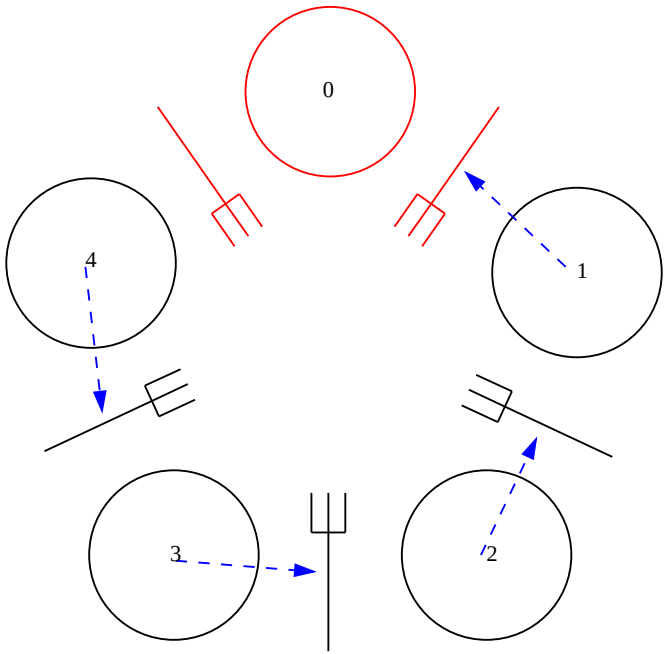
```
parallel philosophers–1
{
    const int P = 5;                          // Anzahl Philosophen
    Semaphore forks[P] = { 1 [P] };           // Gabeln

    process Philosopher [int p ∈ {0, ..., P − 1}] {
        while (1) {
            Denke;
            P(fork[p]);                       // linke Gabel
            P(fork[(p + 1) mod P]);           // rechte Gabel
            Esse;
            V(fork[p]);                       // linke Gabel
            V(fork[(p + 1) mod P]);           // rechte Gabel
        }
    }
}
```

Naive Philosophen

Philosophen sind verklemmt, falls alle zuerst die rechte Gabel nehmen!

Einfache Lösung des Deadlockproblems: Vermeide zyklische Abhängigkeiten, z. B. dadurch, dass der Philosoph 0 seine Gabeln in der anderen Reihenfolge links/rechts nimmt.

Diese Lösung führt eventuell nicht zu maximaler Parallelität:



Schlaue Philosophen

Nehme Gabeln nur wenn *beide* frei sind

Kritischer Abschnitt: nur einer kann Gabeln manipulieren

Drei Zustände eines Philosophen: denkend, hungrig, essend

Programm (Lösung des Philosophenproblems)

```
parallel philosophers–2
{
    const int P = 5;                    // Anzahl Philosophen
    const int think=0, hungry=1, eat=2;
    Semaphore mutex=1;
    Semaphore s[P] = { 0 [P] };         // essender Philosoph
    int state[P] = { think [P] };       // Zustand


}
```

Schlaue Philosophen

Programm (Lösung des Philosophenproblems)

```
parallel process
{
    Philosopher [int p ∈ {0, ..., P − 1}] {
        void test (int i) {
            int l=(i + P − 1) mod P, r=(i + 1) mod P;
            if (state[i]==hungry ∧ state[l]≠eat ∧ state[r]≠eat)
            {
                state[i] = eat;
                V(s[i]);
            }
        }

        while (1) {
            Denke;
            P(mutex);                          // Gabeln nehmen
            state[p] = hungry;
            test(p);
            V(mutex);
            P(s[p]);                           // warte, falls Nachbar isst
            Esse;
            P(mutex);                          // Gabeln weglegen
            state[p] = think;
            test((p + P − 1) mod P);           // wecke l. Nachbarn
            test((p + 1) mod P);               // wecke r. Nachbarn
            V(mutex);
        }
    }
}
```