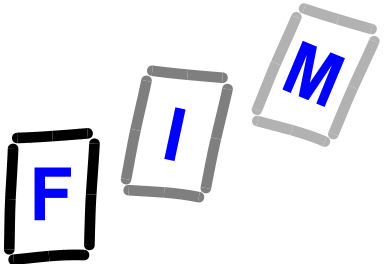
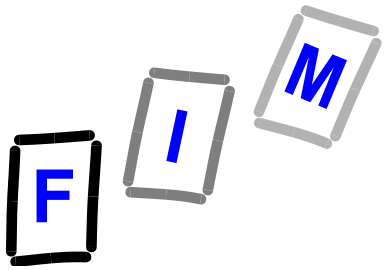


Betriebssysteme

H:

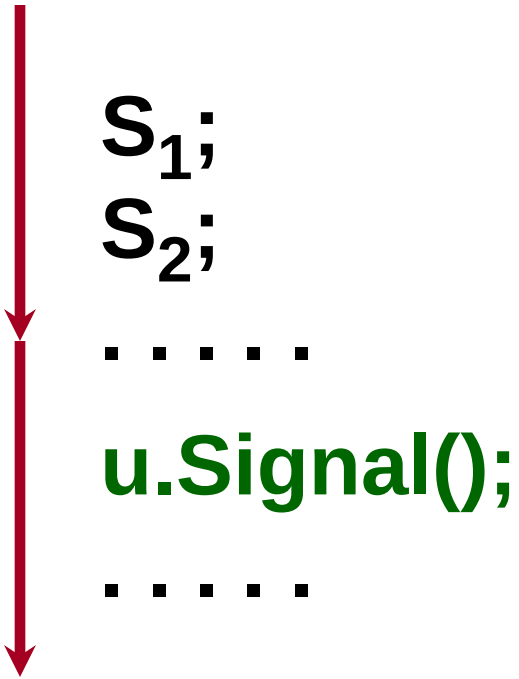
Deadlocks - Systemverklemmungen

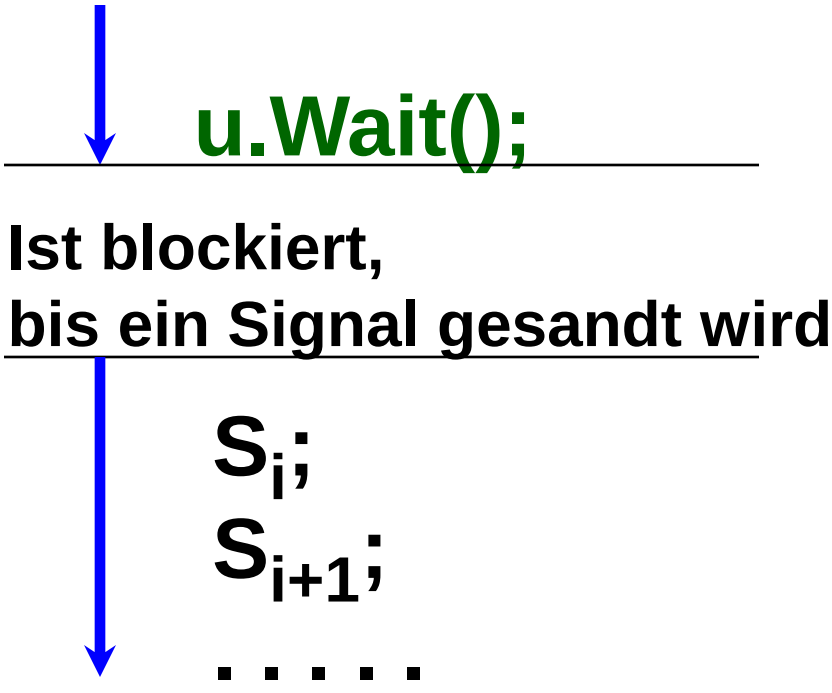
(Teil A: Grundlagen)



Prozess-Synchronisation

- Sei gegeben VAR u:semaphore;
- Initialisiert mit u.Init(0);





"Bibliotheksbeispiel"
Transitionen (informell)

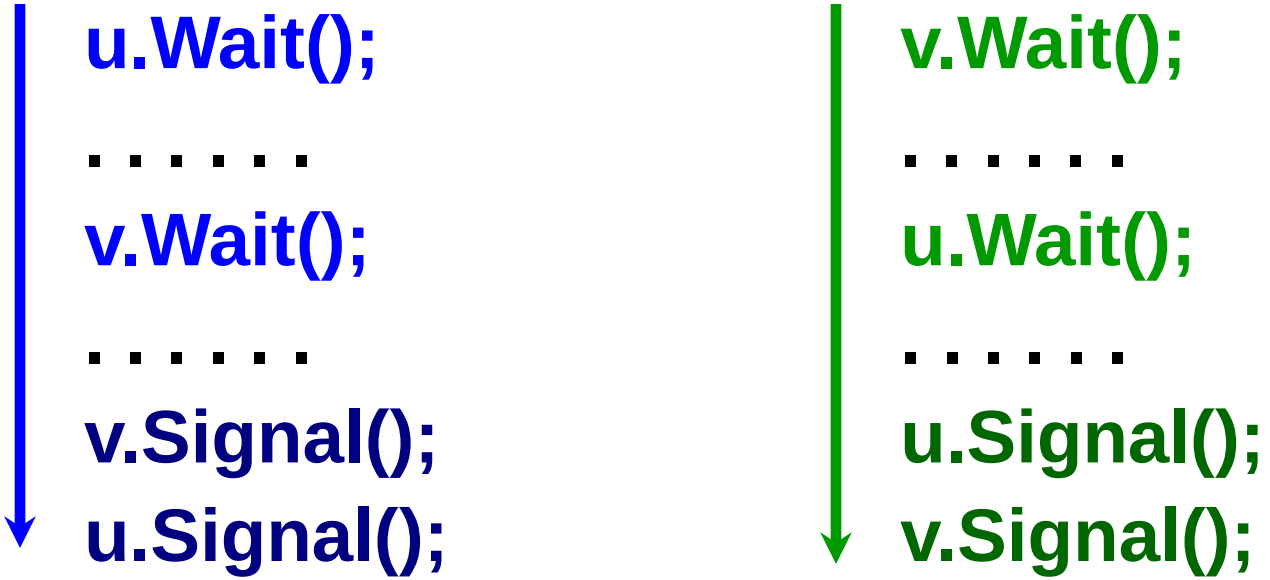
- **In der Bibliothek sind zwei Bücher U und V**
- **Ein Student P1 und eine Studentin P2 wollen jeweils beide entleihen**
- **Zwei Szenarien**

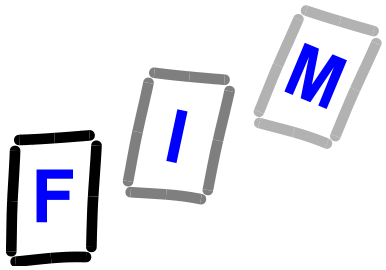
➡ **P1 leiht U und V beide zugleich aus und gibt sie nach endlicher Zeit zurück**
P2 leiht (danach) U und V beide zugleich aus und gibt sie nach endlicher Zeit zurück

➡ **P1 leiht U zuerst aus, weil er V nicht sofort benötigt**
P2 leiht V zuerst aus, weil sie U nicht sofort benötigt
P1 will nun V haben, ohne vorher U zurückzugeben
P2 will nun U haben, ohne vorher V zurückzugeben

Grundproblem (1)

- Sei VAR u,v: semaphore, initialisiert mit INIT(u,1) and INIT(v,1)
- Betrachte die beiden parallelen Prozesse P1 und P2





Grundproblem (2)

- **Sei folgende Verzahnung z.B. gegeben:**

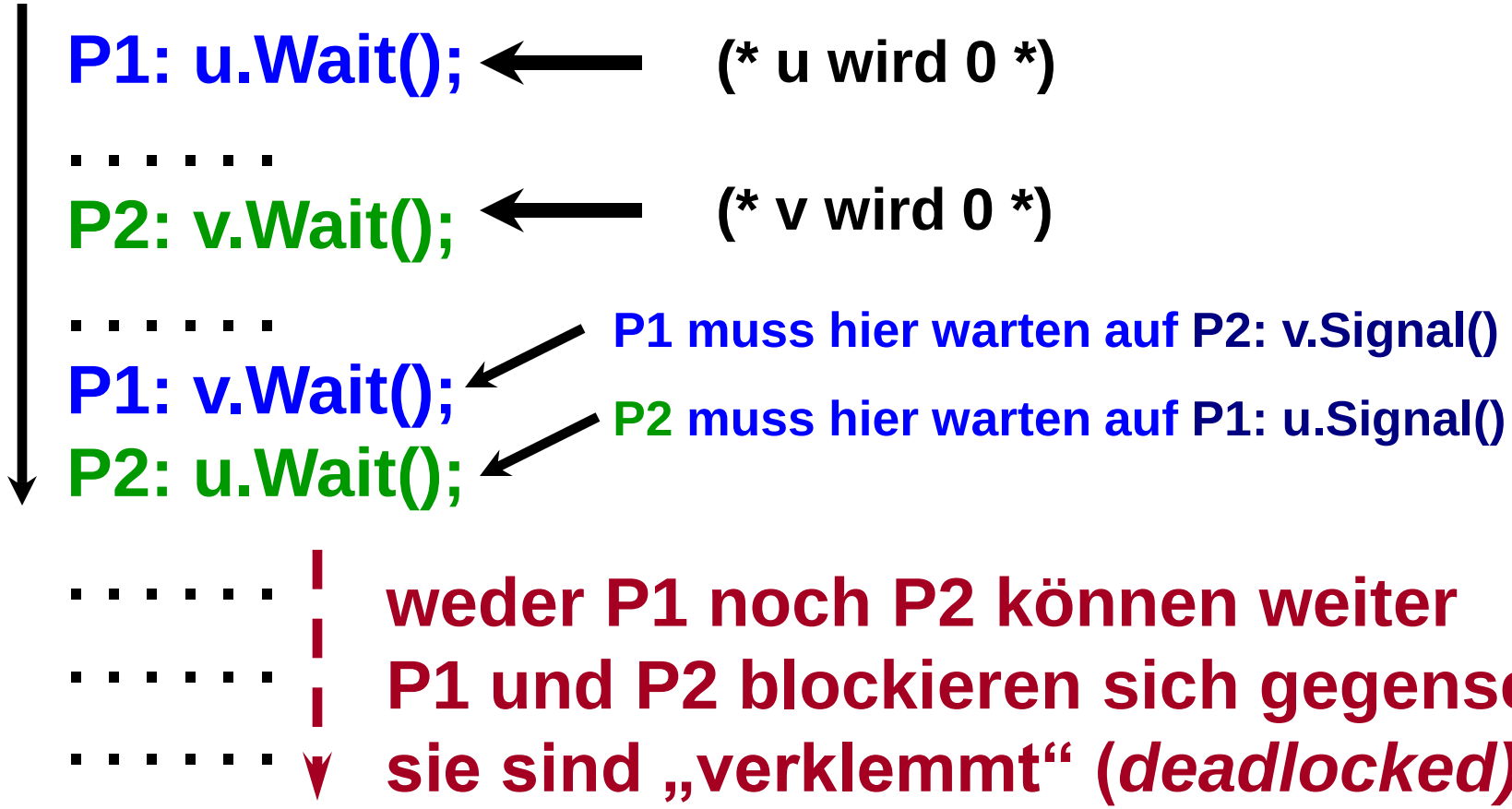
P1: u.Wait();
P1: v.Wait();
P1: v.Signal();
. . . . . .
P2: v.Wait();
. . . . . .
P1: u.Signal();
. . . . . .
P2: u.Wait();
. . . . . .

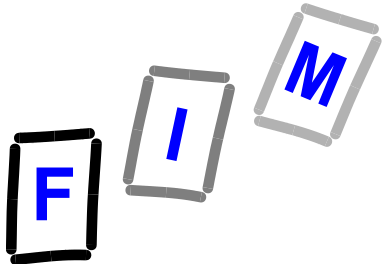
P2 kann weiterlaufen, weil v.Signal() bereits von P1 gesandt wurde

P2 kann weiterlaufen, weil u.Signal() bereits von P1 gesandt wurde

Grundproblem (3)

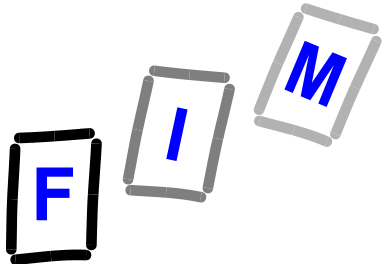
- Jetzt aber betrachte folgende Verzahnung





Deadlock (Definition)

Eine Menge von Prozessen befindet sich in einem „Deadlock“, wenn jeder Prozess aus dieser Menge auf ein Ereignis wartet, das nur von einem anderen Prozess aus derselben Menge ausgelöst werden kann.



Bedingungen (Coffman, 1971)

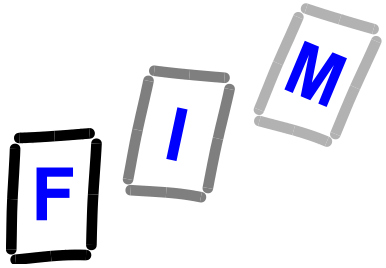
- *(1) Wechselseitiger Ausschluss*

  Mutual Exclusion
- *(2) Halten und Warten*

  Hold and Wait
- *(3) Kein Betriebsmittelentzug*

  No Preemption
- *(4) Zirkulares Warten*

  Circular Wait

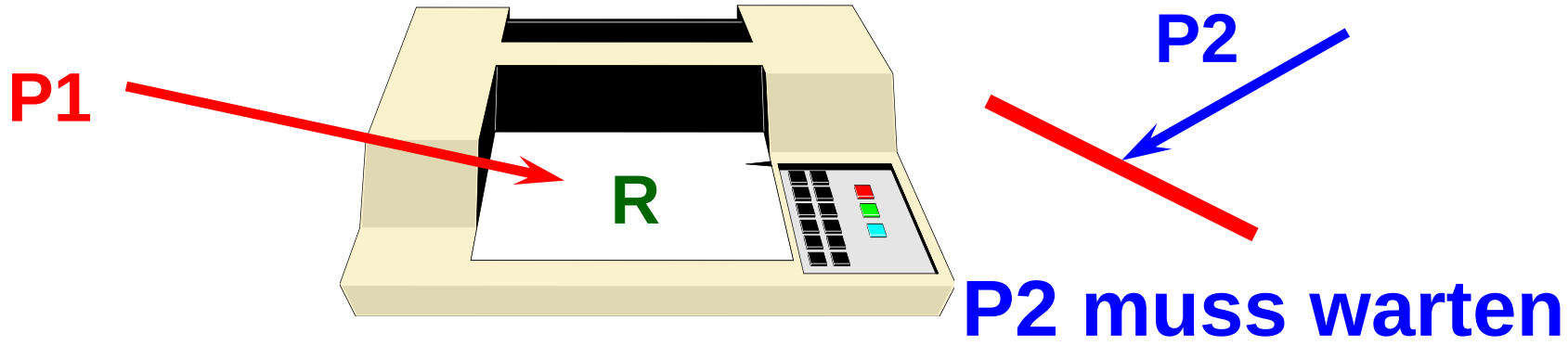


Bedingungen (Coffman, 1971)

- **Notwendig :**
  Bedingungen (1),(2),(3),(4)
- **Hinreichend:**
  Wenn alle 4 Bedingungen *gleichzeitig* halten, ist immer eine Systemverklemmung bereits eingetreten

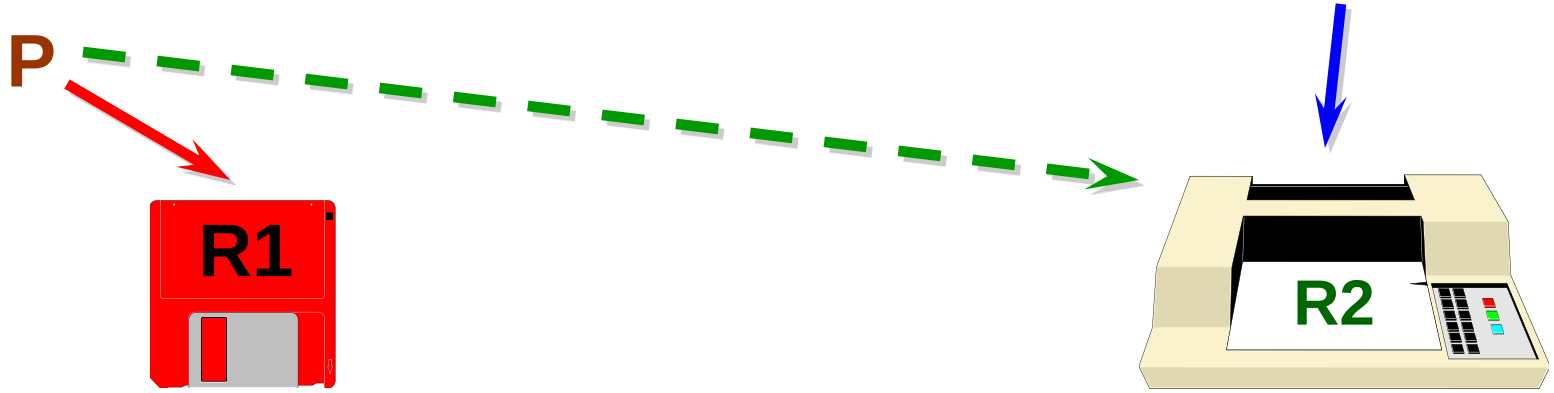
Deadlock Bedingung 1: Mutual Exclusion

- Zu jedem Zeitpunkt kann ein Betriebsmittel R von höchstens einem Prozess P belegt werden
- Jeder andere P, der ebenfalls auf R zugreifen möchte, muss angehalten werden, bis R freigegeben worden ist.



Deadlock Bedingung 2: Hold and Wait

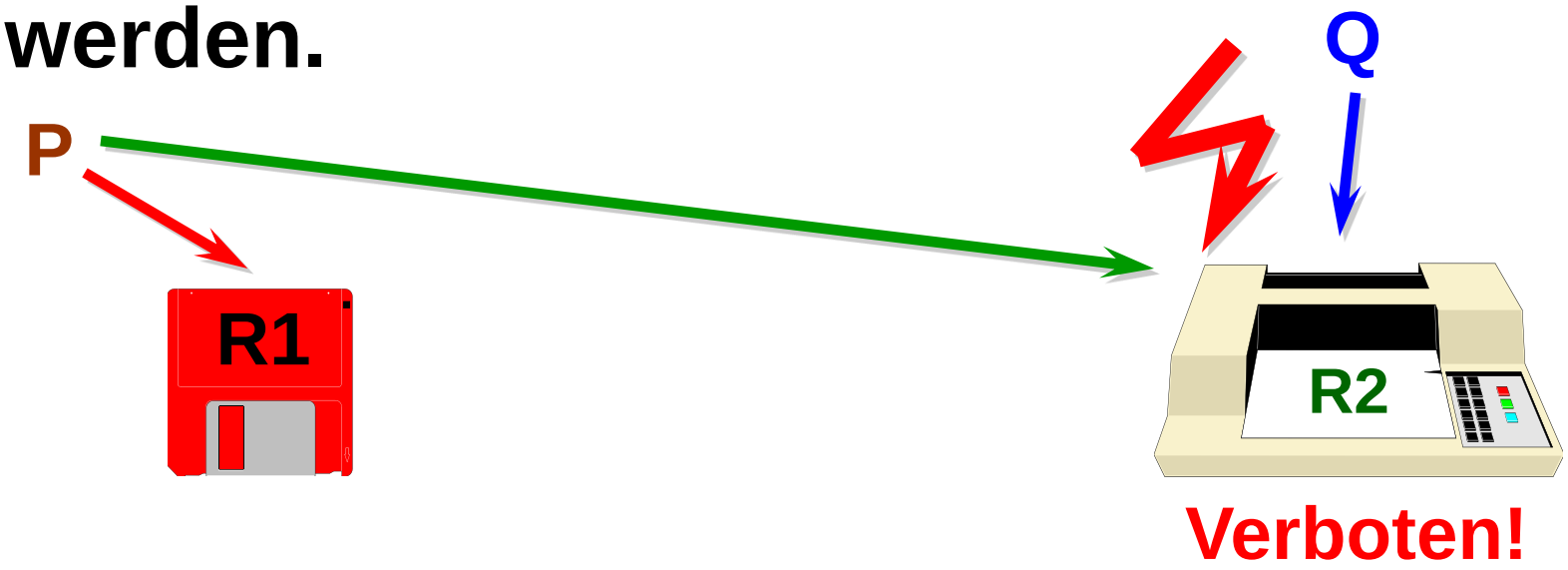
- Es gibt einen P, der mindestens eine Ressource R1 hält und nicht freigibt, während er auf eine Ressource R2 wartet, die momentan von einem anderen Prozess Q belegt wird.

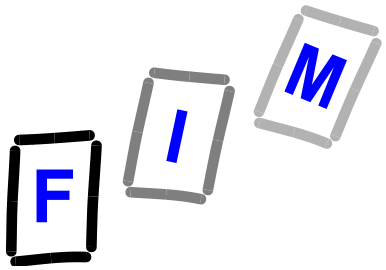


P gibt R1 während des Wartens auf R2 *nicht* frei

Deadlock Bedingung 3: No Preemption

- Ressourcen R_x können nicht entzogen werden: R_x kann nur freiwillig von einem Prozess Q nach Beendigung seiner R betreffenden Aufgabe freigegeben werden.



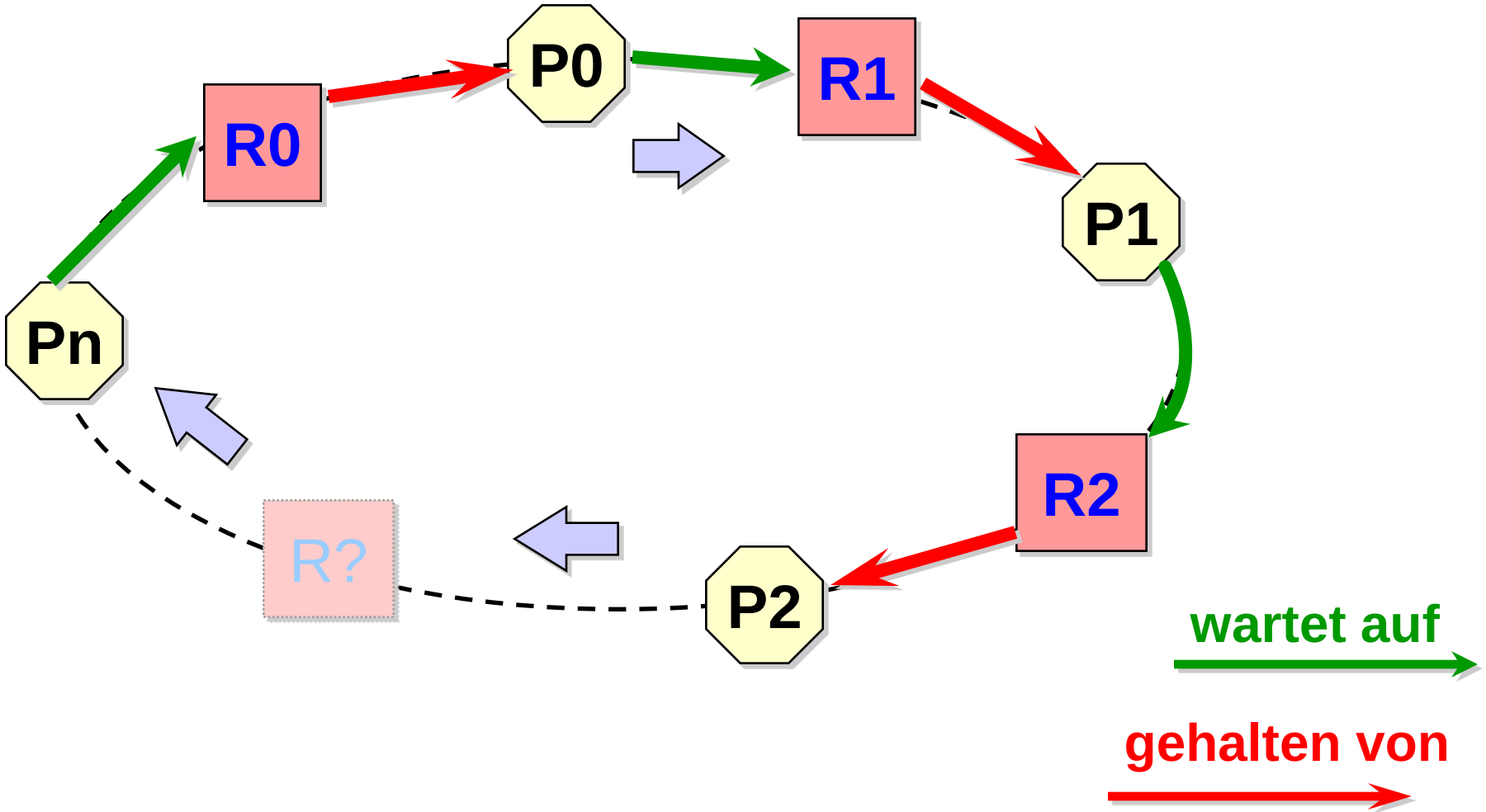


Deadlock Bedingung 4: Circular Wait

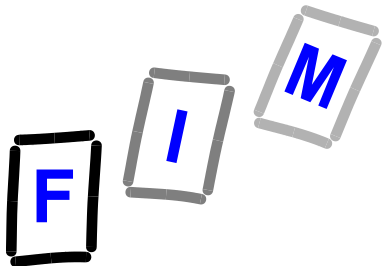
Eine Menge $\{P_0, P_1, P_2, ..., P_n\}$ von wartenden Prozessen existiert, sodass

- P_0 wartet auf eine Ressource R_1, die von P_1 gehalten wird,
- und P_1 wartet auf eine Ressource R_2, die von P_2 gehalten wird,
- und . . . . . .
- und P_n wartet auf eine Ressource R_0, die von P_0 gehalten wird

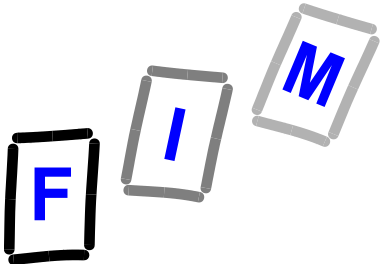
Deadlock Bedingung 4: Circular Wait



Kreisverkehr mit Deadlock



- Man überlege, dass in diesem Beispiel alle 4 Bedingungen halten.
- Man gebe eine einfache Regel an, sodass kein Deadlock eintreten kann.



Die essenden Philosophen

(Klassiker in der Literatur)

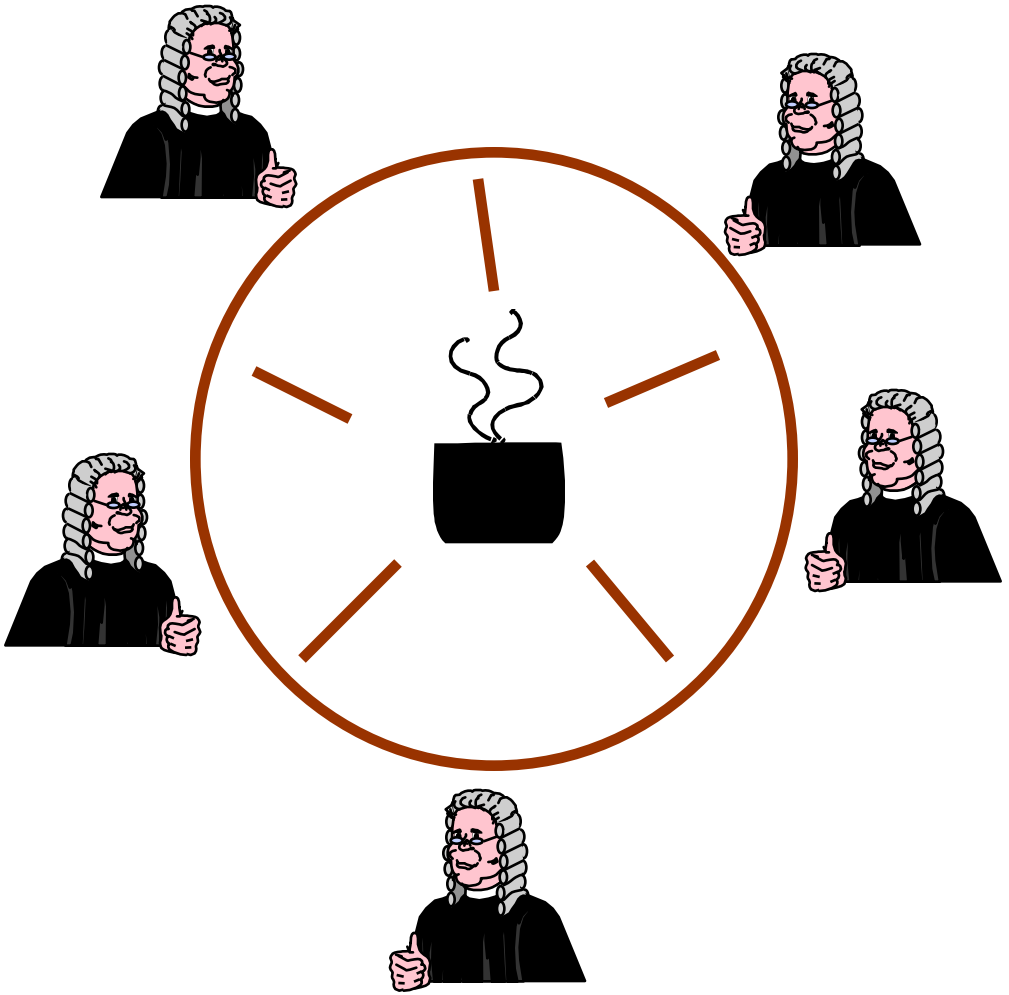
- Betrachte 5 Philosophen Ph_i , die ihr Leben nur mit Denken und zwischenzeitlichem Essen verbringen:

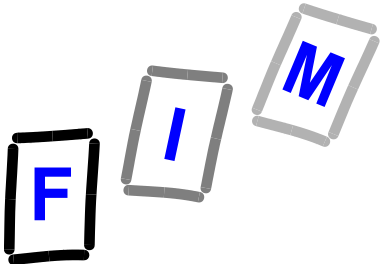
```
do {
    eat(); think();
while(true);
```

- Diese sitzen um einen runden Tisch, auf dem in der Mitte eine Schale mit Reis steht
- Auf dem Tisch sind 5 Ess-Stäbchen (siehe Zeichnung)
- Wenn Ph_i isst, benötigt er 2 Ess-Stäbchen

Die essenden Philosophen

(Klassiker in der Literatur)



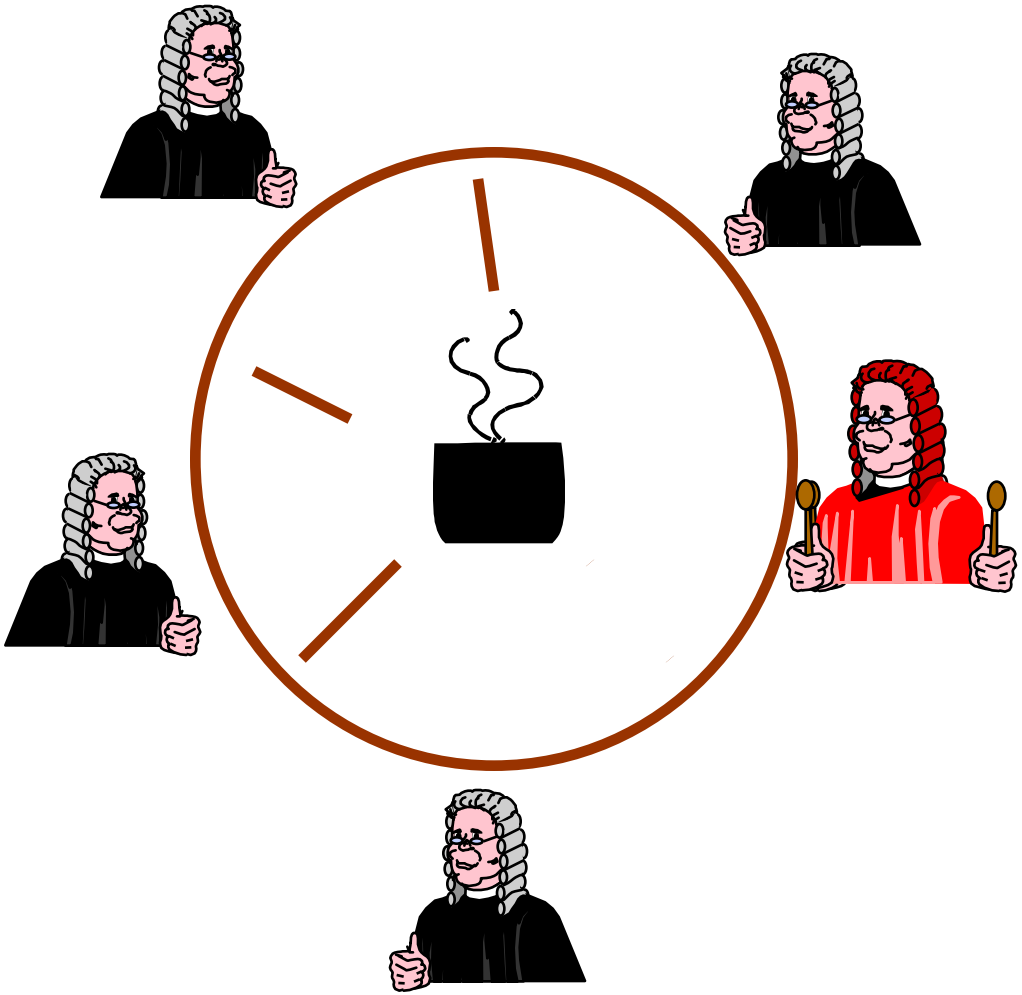


Ein (falscher) Versuch

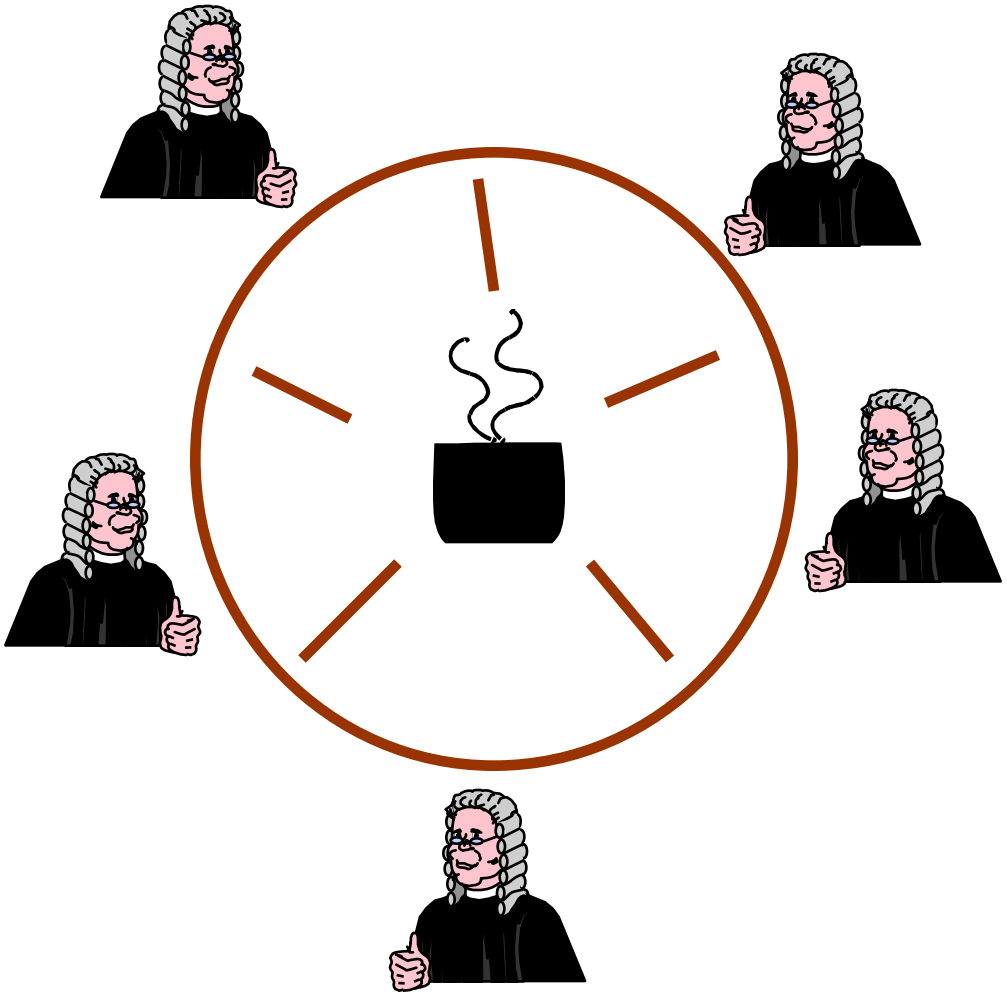
- **Zugriff auf ein Stäbchen kann nur in einer kritischen Region CR erfolgen:**
  Stäbchen können nur unter mutual exclusion benutzt werden: → Verwende Semaphore

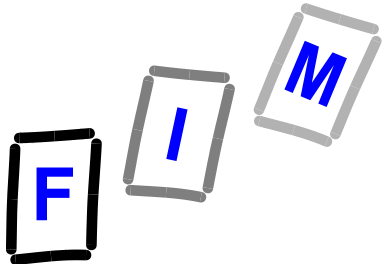
  Semaphore chopstick[] = new Semaphore[5];
- **(z.B.) Ph_3 muss warten,**
  bis beide Stäbchen chopstick[3+1] links und chopstick[3] rechts verfügbar sind
- **Wir adressieren: chopstick[i mod 5]:**
  - ➡ **chopstick[(i +1) mod 5] zur linken Hand**
  - ➡ **chopstick[i mod 5] zur rechten Hand**

Ein Philosoph isst …



… und gibt das Besteck wieder zurück

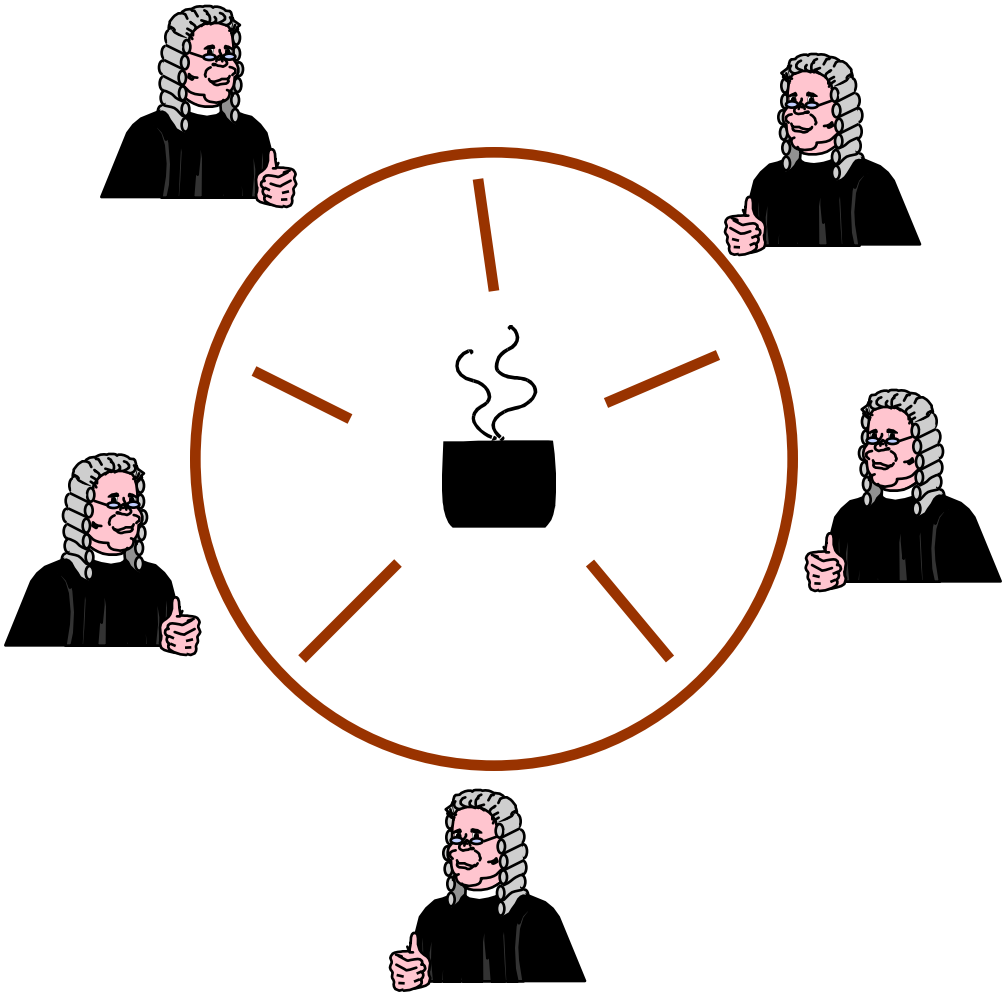




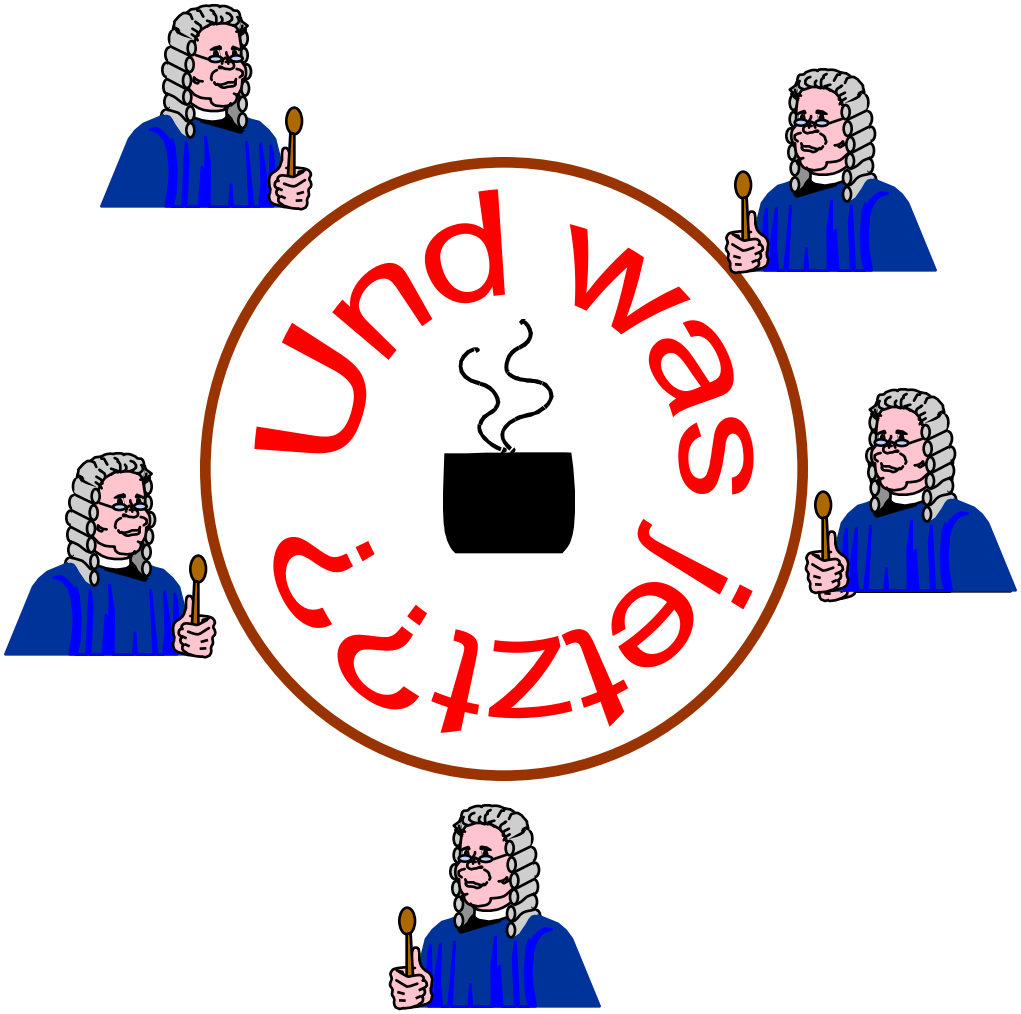
Ein (falscher) Versuch

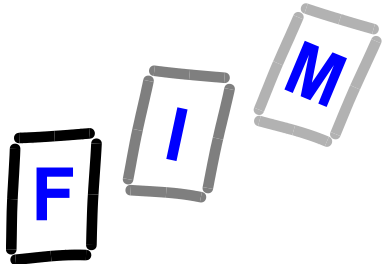
```
initialise chopstick[0..4] with 1 ;

do { (*philosopher PH[i]*)
  chopstick[i].Wait();
  chopstick[(i+1) mod 5]).Wait();
  . . . . . .
  eat();
  chopstick[i].Signal();
  chopstick[(i+1) mod 5].Signal();
  . . . . . .
  think();
} while (true);
```

Ausgangslage



Und nun alle gleichzeitig → Deadlock



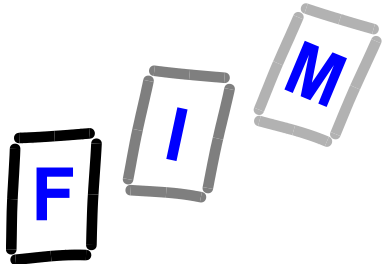


Ein (falscher) Versuch

- **Werden alle Philosophen gleichzeitig hungrig und greifen sie gleichzeitig nach dem jeweils linken Stäbchen, dann sind alle Stäbchen belegt und alle Philosophen warten „unendlich lange“ auf die Freigabe des rechten Stäbchens, also auf**

  chopstick[(i+1) mod 5].Signal();

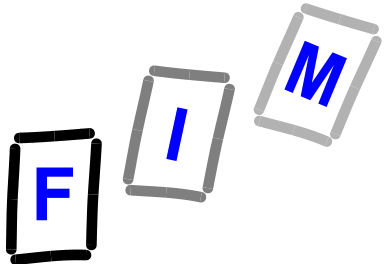
- **Die Philosophen befinden sich ein einer Systemverklemmung und würden verhungern**
- **(Sind „verklemmt“ wäre eine andere sprachliche Bedeutung, die wohl nicht gemeint ist ☺ !)**



Hinweise für eine korrekte Lösung (1)

- **Einem Philosophen sei nur erlaubt, ein Stäbchen (eigentlich: beide) zu nehmen, wenn beide zugleich verfügbar sind. Der Test dazu hat in einer CR zu erfolgen.**

- **Achtung: Es gibt noch eine Vielzahl anderer möglicher Lösungen!**



Hinweise für eine korrekte Lösung (2)

- **Aber ein Problem bleibt!**

  Einer der Philosophen könnte „verhungern“.
- **Lösung durch „aging“ (Priorität abhängig von der Wartezeit)**
- **Der vorige Ansatz gewährleistet zwar Deadlock-Freiheit, beseitigt aber nicht das Problem „starvation“!**