Prozesssynchronisation II

Kurze Auffrischung

In der vorangegangenen Einheit haben Sie gelernt:

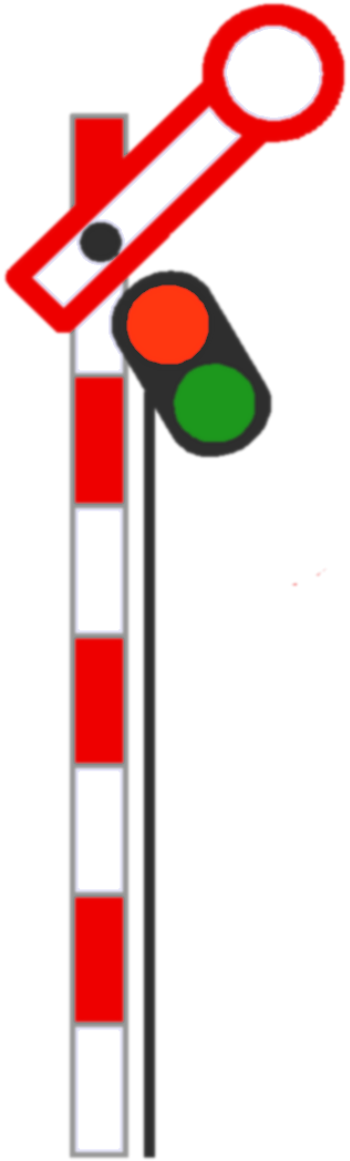
- **konkurrente Prozesse**, die gemeinsame Daten oder Betriebsmittel nutzen wollen, müssen ggf. **synchronisiert** werden.
- Als wichtigstes Synchronisationsmittel wird **gegenseitiger Ausschluss** (*mutex*) für kritische Abschnitte benötigt.
- Softwarelösungen im User-Mode sind möglich, jedoch
  - sind sie **aufwändig** und **fehleranfällig**,
  - verwenden sie stets **busy waits**.

Semaphoren

Es ist sinnvoll, wenn das **Betriebssystem** generische Funktionen zur Synchronisation von Prozessen bereitstellt. Dies senkt nicht nur die Fehleranfälligkeit, sondern kann auch *busy waits* vermeiden, da das Betriebssystem wartende Prozesse **vom Scheduling ausschließen** (blockieren) kann. So können diejenigen Prozesse, die sich im Zustand „bereit“ befinden, die CPU verwenden.

Eine mögliche und weit verbreitete Lösung sind **Semaphoren** (griech.: *Zeichenträger*, *Signalbake*), die von Dijkstra 1965 vorgeschlagen wurden.

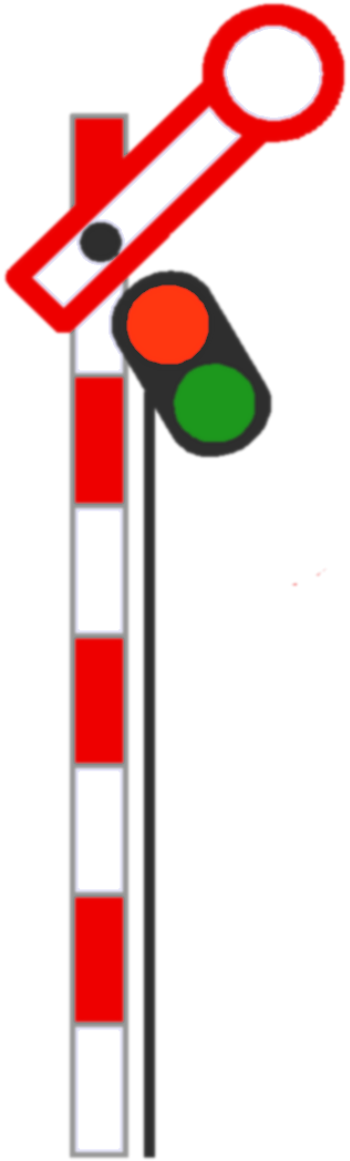
Semaphoren – Eigenschaften



Ein Semaphor ist eine vom Betriebssystem bereitgestellte Schnittstelle zur Prozess-synchronisation. Er kann als Signal interpretiert werden, an dem in bestimmten Fällen gewartet werden muss. Der Semaphor beinhaltet

1. einen **Zähler**, der angibt, wie viele Prozesse **aktuell** noch den Semaphor passieren dürfen. In der bisherigen Terminologie entspricht dies der Anzahl der Prozesse, die den betreffenden kritischen Abschnitt aktuell noch betreten dürfen.

Semaphoren – Eigenschaften

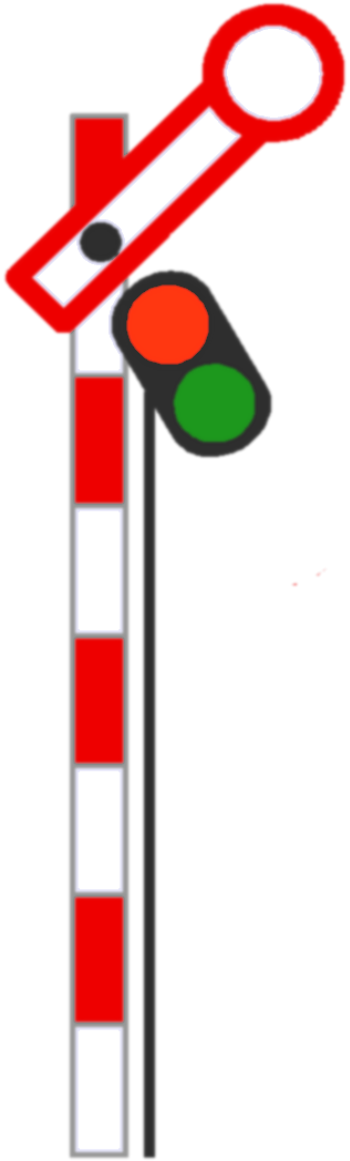


2. die **Funktionen** *Passieren* und *Verlassen*, mit denen der Wert des Semaphors geprüft und ggf. verändert wird,

3. eine **Warteschlange**, in der Prozesse, die den Semaphor bisher nicht passieren konnten, verwaltet werden.

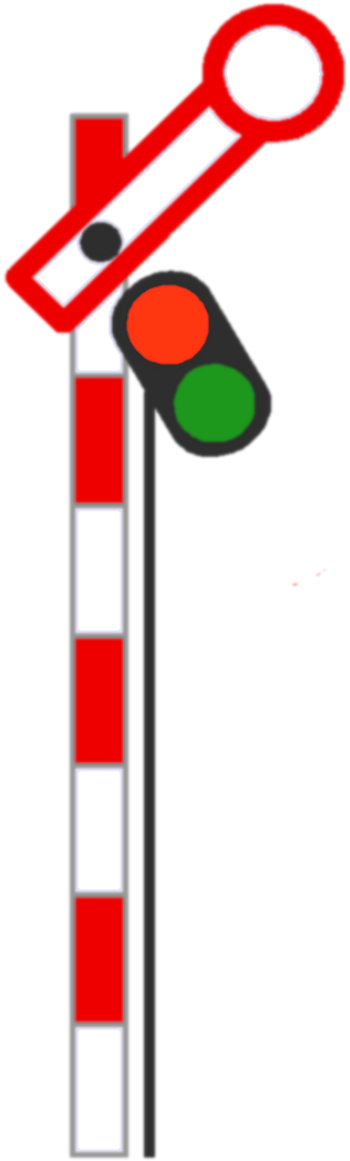
Passieren und *Verlassen* sind als **atomare Systemaufrufe** implementiert, können also nicht unterbrochen werden. Dies ist für das Betriebssystem leicht umsetzbar, weil es im Kernelmodus eventuell auftretende Interrupts sperren darf.

Semaphoren – Eigenschaften



Der **Startwert** des Zählers gibt an, wie viele **Prozesse maximal gleichzeitig** den durch den Semaphor geschützten kritischen Abschnitt betreten dürfen. Bei gegenseitigem Ausschluss hat er den Wert 1.

Semaphor als Datentyp

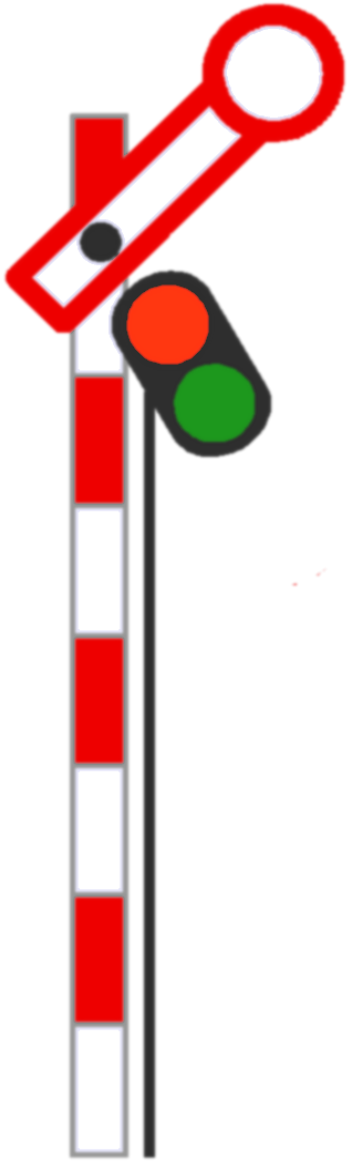


Der **Datentyp** Semaphor lässt sich also in Pseudocode in etwa beschreiben als:

```
Semaphor {
    wert:   integer;
    liste: List <Prozess>;
}

function P(Semaphor s, Prozess p);
function V(Semaphor s);
```

Semaphoren – Eigenschaften



Jeder Prozess, der den durch den Semaphor geschützten kritischen Abschnitt **betreten** will, muss dies durch den Aufruf der Funktion ***Passieren*** ankündigen. Da es sich um einen Systemruf handelt, übernimmt das Betriebssystem kurzzeitig die Kontrolle und prüft, ob der Zutritt möglich ist.

Beim Beenden des kritischen Abschnitts muss der Prozess die Funktion ***Verlassen*** aufrufen. Auch dies ist ein Systemaufruf.

Die Systemrufe *Passieren* und *Verlassen*

Passieren P(s): Prüfen, ob interner Zähler > 0

- **Wenn ja:** Zähler um 1 dekrementieren, der aufrufende Prozess darf den kritischen Abschnitt betreten.
- **Wenn nein:** Prozess in den Zustand „blockiert“ versetzen und in **Warteschlange** einreihen.

```
Prozess p;      // Anm.: Der aufrufende Prozess

Funktion P(Semaphor s) {
  Ist s.wert = 0?
    Ja:     einhängen (p, s.liste);
            p.status := BLOCKIERT;
    Nein:   dekrementiere (s.wert);
}
```

Eigenschaften der Funktionen *Passieren* und *Verlassen*

Verlassen V(s): Prüfen, ob weitere Prozesse auf den Zutritt zum kritischen Abschnitt warten (also beim Passieren blockiert wurden).

- **Wenn ja:** Nächsten blockierten Prozesses aus der Warteschlange entnehmen und dessen Zustand wieder auf „bereit“ setzen.
- **Wenn nein:** Zähler um 1 inkrementieren.

```
Funktion V(Semaphor s) {
  Ist s.liste.laenge > 0?
    Ja:     Prozess p := liste.erster;
            p.status := BEREIT;
    Nein:   inkrementiere (s.wert);
}
```

Beispiel gegenseitiger Ausschluss

Die Implementierung des gegenseitigen Ausschlusses ist nun sehr einfach möglich, es wird nur ein einziger Semaphor benötigt:

Initialisierung:

```
s: Semaphor;
s.wert := 1;
```

Prozess 1

```
P(s)
kritische Operationen
V(s)
```

Prozess 2

```
P(s)
kritische Operationen
V(s)
```

Beispiel gegenseitiger Ausschluss

Zur Verdeutlichung wird nun ein möglicher Ablauf mit Zeotscheibenverfahren schrittweise beschrieben.

Prozess 1	**Prozess 2**

Beispiel gegenseitiger Ausschluss – Analyse

Initialisierung:

1. Semaphore wird erzeugt, `s.wert` ist 1, `liste` ist leer.

```
s.wert := 1;
```

```
liste := {};
```

Prozess 1

Prozess 2

Beispiel gegenseitiger Ausschluss – Analyse

Annahme: Prozess 1 erhält zuerst die CPU.

2. Prozess 1 ruft `P(s)` auf

```
s.wert := 1;
```

```
liste := {};
```

Prozess 1

```
P(s)
```

Prozess 2

Beispiel gegenseitiger Ausschluss – Analyse

Annahme: Prozess 1 erhält zuerst die CPU.

2. Prozess 1 ruft `P(s)` auf
3. Da `s.wert` > 0, kann Prozess 1 passieren und `s.wert` wird um 1 heruntergezählt.

```
s.wert := 0;
```

```
liste := {};
```

Prozess 1

```
P(s)
```

Prozess 2

Beispiel gegenseitiger Ausschluss – Analyse

Nun ist zufällig die Zeitscheibe abgelaufen, Prozess 2 startet.

4. Prozess 2 ruft ebenfalls P(s) auf

```
s.wert := 0;
```

```
liste := {};
```

Prozess 1

```
P(s)
```

Prozess 2

```
P(s)
```

Beispiel gegenseitiger Ausschluss – Analyse

Nun ist zufällig die Zeitscheibe abgelaufen, Prozess 2 startet.

4. Prozess 2 ruft ebenfalls P(s) auf
5. Da `s.wert` = 0, kann Prozess 2 **nicht** passieren und wird in die Warteliste eingefügt. `s.wert` bleibt 0.

```
s.wert := 0;
```

```
liste := {Prozess 2};
```

Prozess 1

```
P(s)
```

Prozess 2

```
P(s) → blockiert
```

Beispiel gegenseitiger Ausschluss – Analyse

Mit der Blockade ist Prozess 2 aus dem Scheduling genommen.

6. Prozess 1 erhält die CPU und betritt den kritischen Abschnitt.

```
s.wert := 0;
```

```
liste := {Prozess 2};
```

Prozess 1

```
P(s)
```

```
kritische Operationen
```

Prozess 2

```
P(s) → blockiert
```

Beispiel gegenseitiger Ausschluss – Analyse

7. Nach Beenden des kritischen Abschnitts ruft Prozess 2 die Funktion `V(s)` auf.

```
s.wert := 0;
```

```
liste := {Prozess 2};
```

Prozess 1

```
P(s)
```

```
kritische Operationen
```

```
V(s)
```

Prozess 2

```
P(s) → blockiert
```

Beispiel gegenseitiger Ausschluss – Analyse

7. Nach Beenden des kritischen Abschnitts ruft Prozess 2 die Funktion `V(s)` auf.

Da in der Warteliste noch Prozesse sind, wird aus dieser der Prozess am Anfang der Liste entnommen und „bereit“ gesetzt.

```
s.wert := 0;
liste := {};
```

Prozess 1

```
P(s)
kritische Operationen
V(s)
```

Prozess 2

```
P(s) → bereit
```

Beispiel gegenseitiger Ausschluss – Analyse

Noch aber hat Prozess 1 die CPU und könnte von vorne beginnen!

```
s.wert := 0;
liste := {};
```

Prozess 1

```
P(s)
kritische Operationen
V(s)
```

Prozess 2

```
P(s)
```

Beispiel gegenseitiger Ausschluss – Analyse

Noch aber hat Prozess 1 die CPU und könnte von vorne beginnen! Zufällig ist nun aber dessen Zeitscheibe abgelaufen, Prozess 2 erhält die CPU.

8. Prozess 2 setzt die Ausführung **hinter** `P(s)` fort ….

```
s.wert := 0;
liste := {};
```

Prozess 1

```
P(s)
kritische Operationen
V(s)
```

Prozess 2

```
P(s)
```

Beispiel gegenseitiger Ausschluss – Analyse

9. … und betritt den kritischen Abschnitt.

```
s.wert := 0;
liste := {};
```

Prozess 1

```
P(s)
kritische Operationen
V(s)
```

Prozess 2

```
P(s)
kritische Operationen
```

Beispiel gegenseitiger Ausschluss – Analyse

9. … und betritt den kritischen Abschnitt.
10. Am Ende ruft Prozess 2 ebenfalls `V(s)` auf.

```
s.wert := 0;
liste := {};
```

Prozess 1

```
P(s)
kritische Operationen
V(s)
```

Prozess 2

```
P(s)
kritische Operationen
V(s)
```

Beispiel gegenseitiger Ausschluss – Analyse

9. … und betritt den kritischen Abschnitt.
10. Am Ende ruft Prozess 2 ebenfalls `V(s)` auf.

Da die **Warteliste leer** ist, wird nun `s.wert` hochgezählt.

```
s.wert := 1;
liste := {};
```

Prozess 1

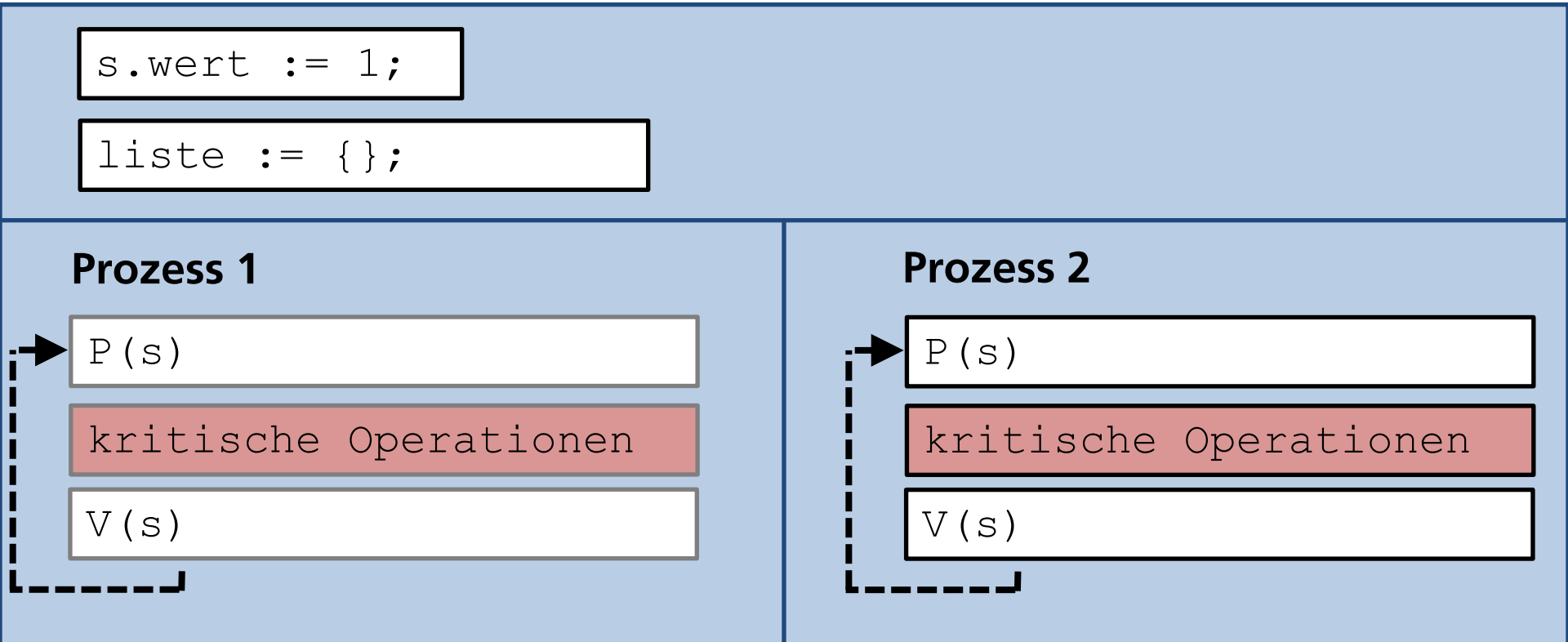
```
P(s)
kritische Operationen
V(s)
```

Prozess 2

```
P(s)
kritische Operationen
V(s)
```

Beispiel gegenseitiger Ausschluss – Analyse

Der Semaphor hat wieder denselben Zustand wie zu Beginn, das Prozedere kann von Neuem beginnen.



Semaphore – Ununterbrechbarkeit

Zur Erinnerung: Die Funktion P und V **müssen** atomar sein, d.h. ihre Bearbeitung darf nicht unterbrochen werden.

Auf **Multiprozessorsystemen** muss zusätzlich sichergestellt werden, dass nicht ein weiterer Prozess **gleichzeitig** denselben Semaphor durch Aufruf von P oder V manipuliert.

Lösung: Hier muss man auf *busy wait* beim Eintritt in P und V zurückgreifen, es gibt keine andere Lösung.

Aber: P und V sind nur einige Instruktionen lang, die Wartezeit ist also sowohl kurz (einige µs) als auch nach oben beschränkt. Dies unterscheidet sich erheblich von aktivem Warten auf den Zugang zu kritischen Abschnitten. Hier ist die Wartezeit ggf. lang, problemabhangig und nicht beschränkt.

Prozesssynchronisation

- Das vorangegangene Beispiel zeigt, wie ein einzelner Semaphor einen kritischen Abschnitt sichern kann.
- Das Anwendungsgebiet für Semaphore ist jedoch breiter. Mit Hilfe von Semaphoren lässt sich bspw.
  1. der Zugriff auf Ressourcen mit begrenzten Kapazitäten steuern (Erzeuger-Verbraucher-Problem),
  2. die Ausführungsreihenfolge nebenläufiger Prozesse festlegen.
- Beide Anwendungsfälle werden nun erläutert.

Das Erzeuger-Verbraucher-Problem

Mit dem **Erzeuger-Verbraucher-Problem** wird ein Alltagsszenario nebenläufiger Prozesse beschrieben:

- Es gibt (mindestens) zwei Prozesse, die über einen gemeinsamen **endlichen Datenpuffer** mit **N** belegbaren **Plätzen** Daten übertragen wollen.
- Der **Erzeuger-Prozess** legt Daten in dem Puffer ab, der **Verbraucher-Prozess** kann diese aus dem Puffer entnehmen.
- Ist der Puffer **leer**, muss der **Verbraucher** warten, bis neue Daten erzeugt werden.
- Ist der Puffer **voll**, muss der **Erzeuger** warten, bis Daten verbraucht wurden.

Das Erzeuger-Verbraucher-Problem

Lösung mit Hilfe von Semaphoren:

- Da Erzeuger und Verbraucher in jeweils **unterschiedlichen** Situationen warten müssen, wird für jede dieser Situationen ein eigener Semaphor benötigt:
  - Für den Erzeuger wird ein Semaphor `frei` verwendet, dessen Wert mit N initialisiert wird.
  - Der Verbraucher erhält einen Semaphor `belegt`, der mit dem Wert 0 initialisiert wird.
- Zusätzlich wird das Schreiben und Lesen des Puffers als **kritischer Abschnitt** mit einem Semaphor `mutex` zum gegenseitigen Ausschluss abgesichert.

Implementierung

Initialisierung:

```
frei, belegt, mutex: Semaphor;
```

```
frei.wert := N;
```

```
belegt.wert := 0;
```

```
mutex.wert := 1;
```

Erzeuger

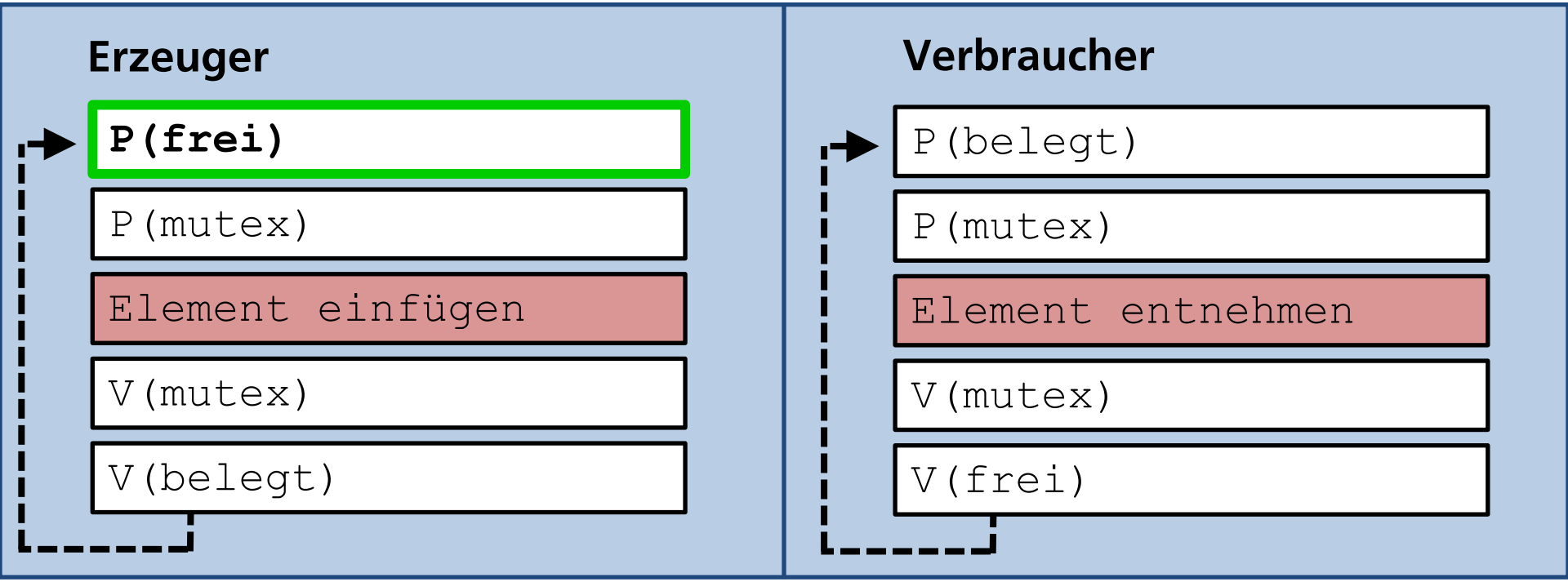
```
P(frei)
P(mutex)
Element einfügen
V(mutex)
V(belegt)
```

Verbraucher

```
P(belegt)
P(mutex)
Element entnehmen
V(mutex)
V(frei)
```

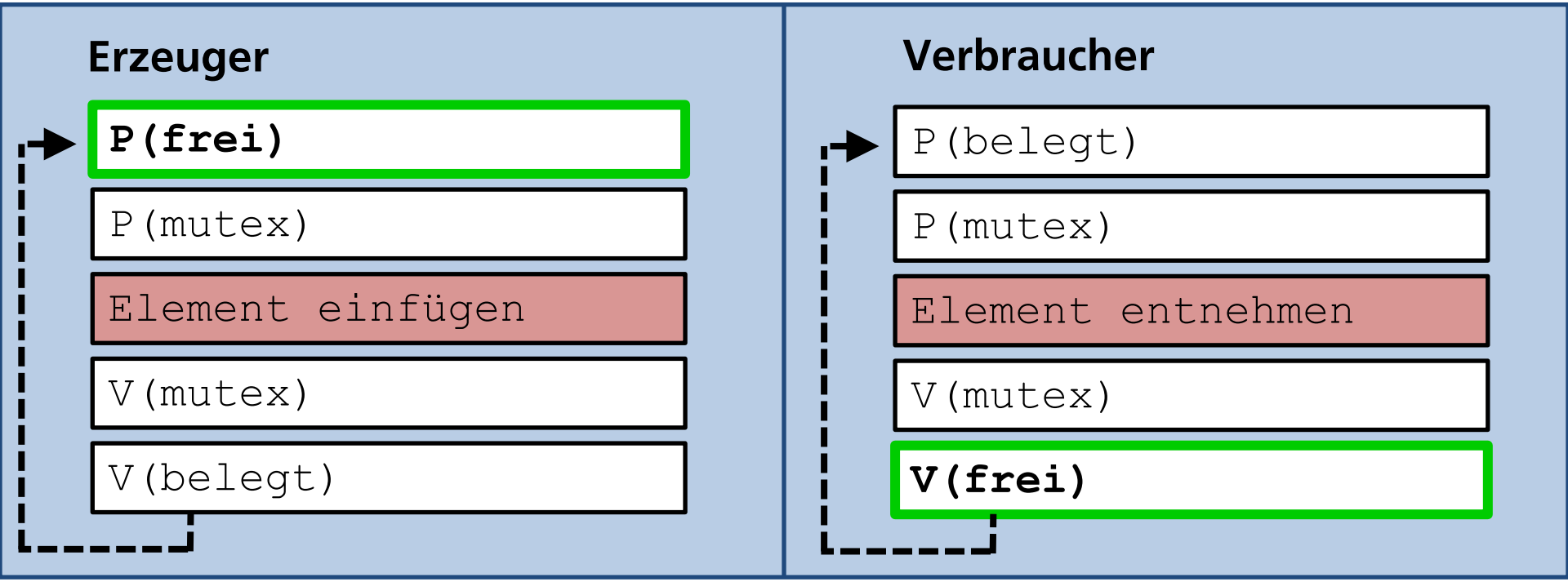
Implementierung – Analyse aus Erzeugersicht

Solange noch Plätze frei sind, ist für den Erzeuger das Passieren des Semaphors `frei` möglich. Mit jeder Passage wird der Wert des Semaphors dekrementiert.



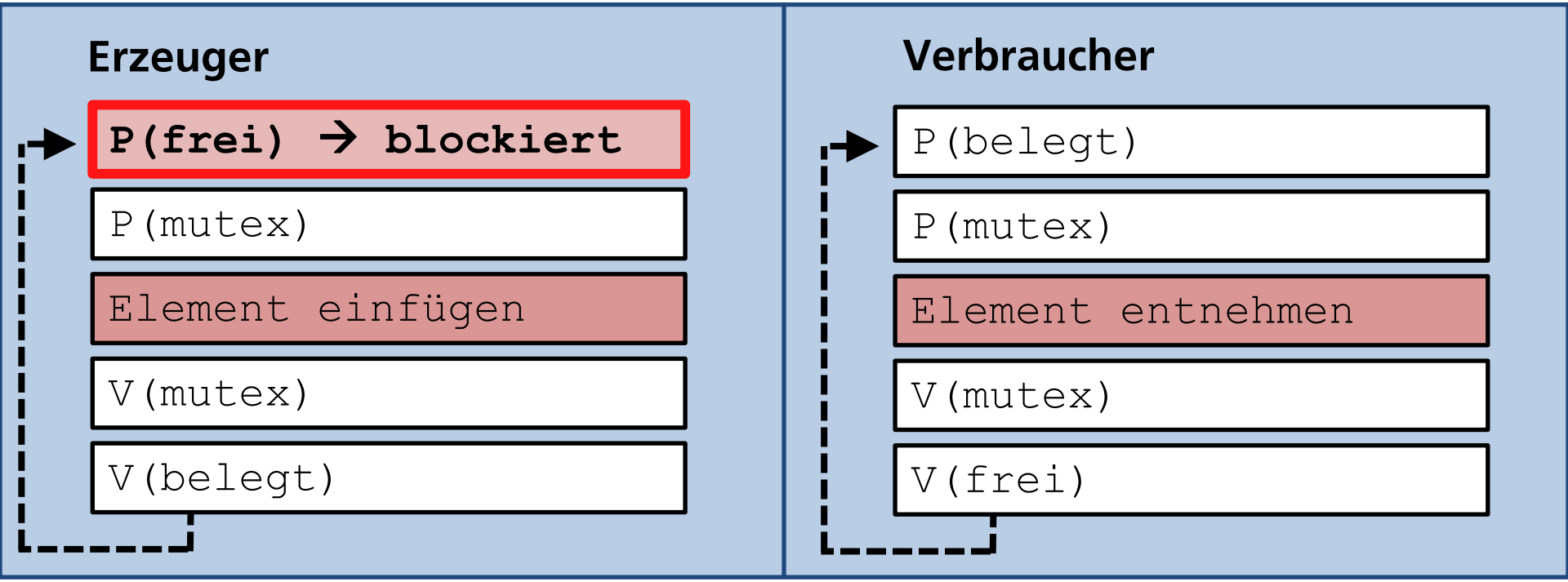
Implementierung – Analyse aus Erzeugersicht

Mit jeder erfolgreichen Entnahme eines Elements **inkrementiert** der Verbraucher diesen Zähler wieder. Solange also der Verbraucher ausreichend schnell Daten entnimmt, kann der Erzeuger ohne zu warten neue Daten einfügen.



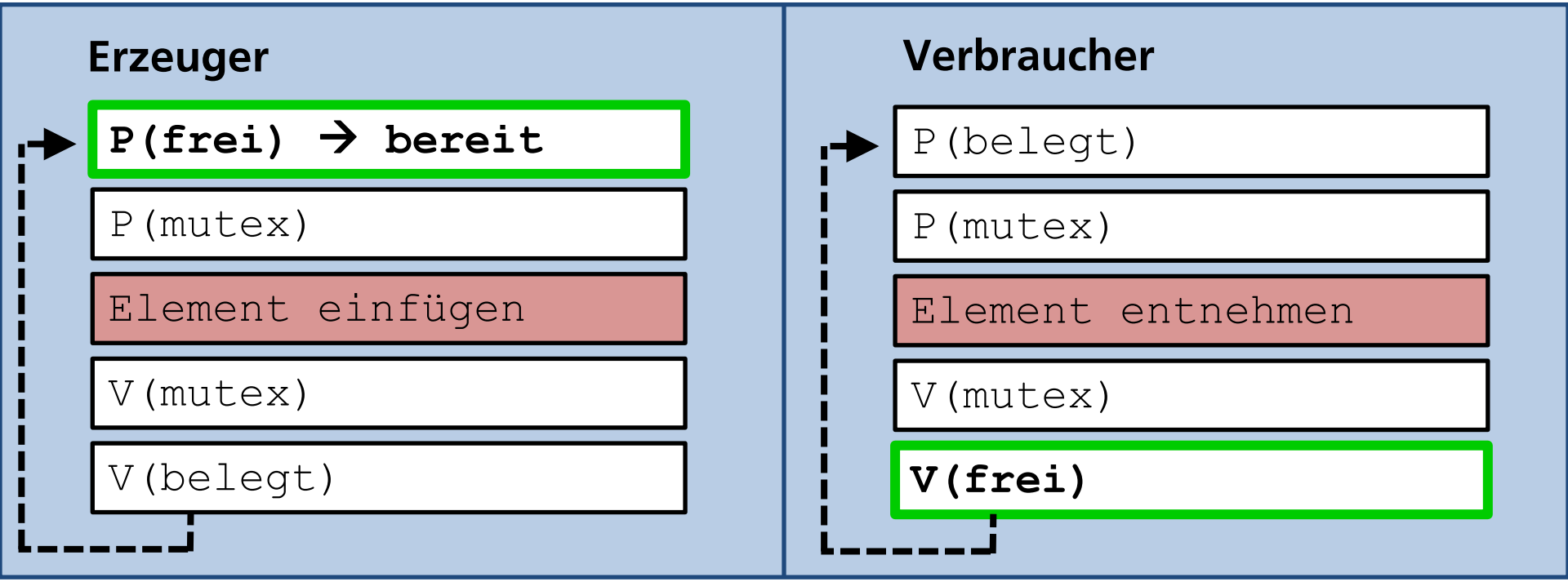
Implementierung – Analyse aus Erzeugersicht

Ist der Datenpuffer voll (bspw. weil der Verbraucher mehr Zeit zur Verarbeitung der Daten benötigt, als der Erzeuger zum Erstellen), kann der Erzeuger an ebendieser Stelle nicht passieren und wird blockiert.



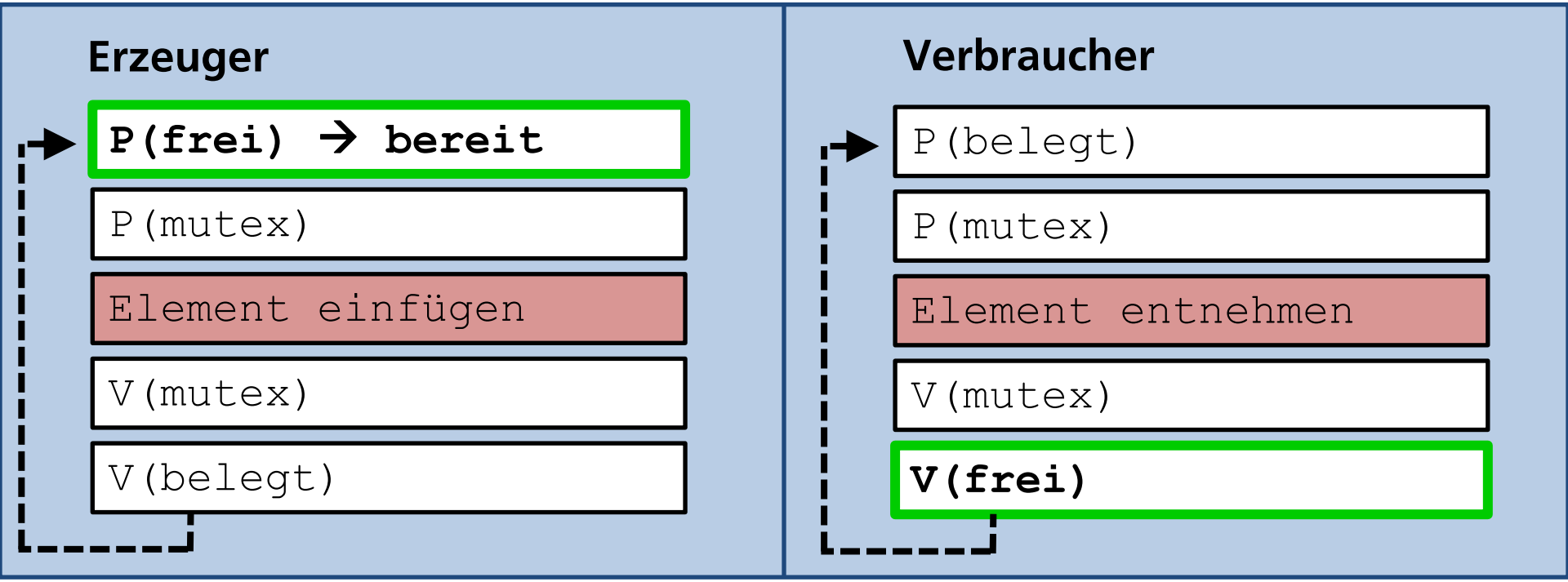
Implementierung – Analyse aus Erzeugersicht

Erst dann, wenn der Verbraucher ein Element aus dem Puffer entnommen hat und `V(frei)` aufruft, kann der Erzeuger passieren und seine Ausführung hinter `P(frei)` fortsetzen.



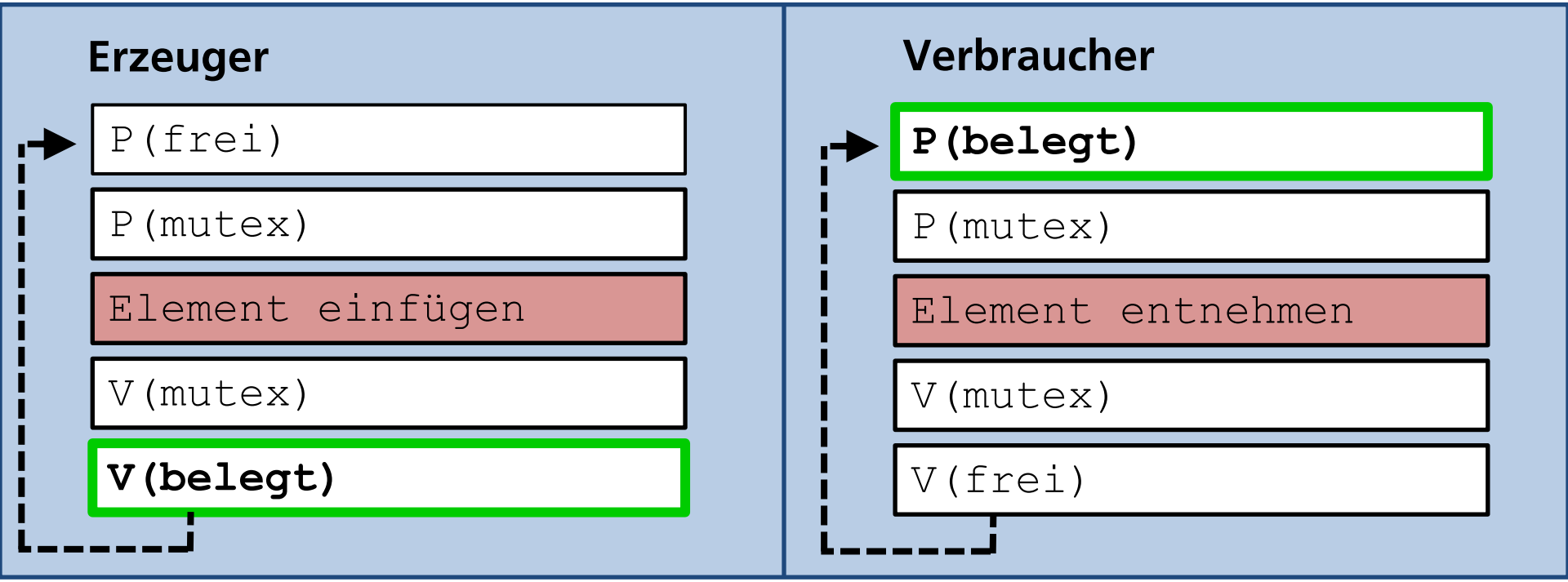
Implementierung – Analyse aus Erzeugersicht

Zur Erinnerung: Der Wert des Semaphors wird nicht erhöht, wenn noch mindestens ein Prozess auf die Passage wartet. Im konkreten Fall bedeutet das: Es wäre zwar kurzzeitig ein Platz im Puffer frei, dieser wird aber sofort vom Erzeuger reserviert.



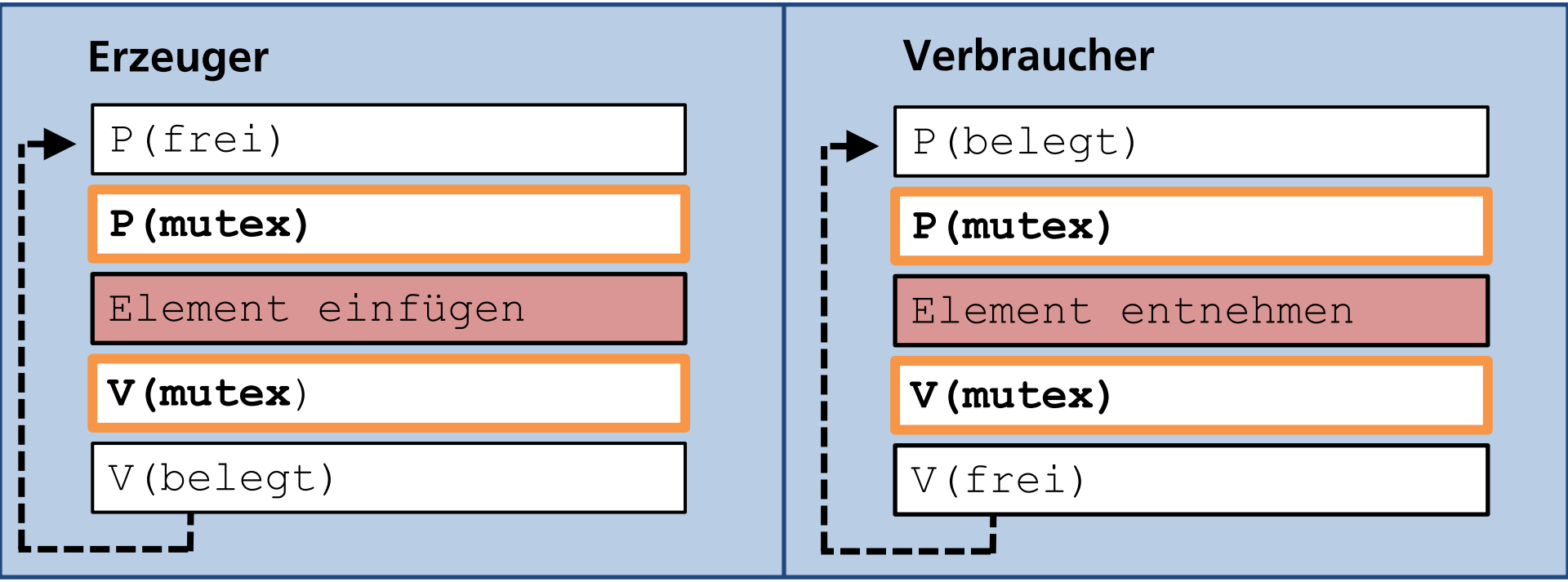
Implementierung – Analyse aus Verbrauchersicht

Analog zum Erzeuger muss der Verbraucher den Semaphor `belegt` passieren, dessen Wert wiederum vom Erzeuger inkrementiert wird.



Implementierung – Analyse aus Verbrauchersicht

Der gegenseitige Ausschluss beim Zugriff auf den Puffer ist notwendig, weil beide Prozesse den Puffer **verändern**.



Prozesssynchronisation

- Das vorangegangene Beispiel zeigt, wie ein einzelner Semaphor einen kritischen Abschnitt sichern kann.
- Das Anwendungsgebiet für Semaphore ist jedoch breiter. Mit Hilfe von Semaphoren lässt sich bspw.
  1. der Zugriff auf Ressourcen mit begrenzten Kapazitäten steuern (Erzeuger-Verbraucher-Problem),
  2. die Ausführungsreihenfolge nebenläufiger Prozesse festlegen.
- Beide Anwendungsfällte werden nun erläutert.

Prozesssynchronisation

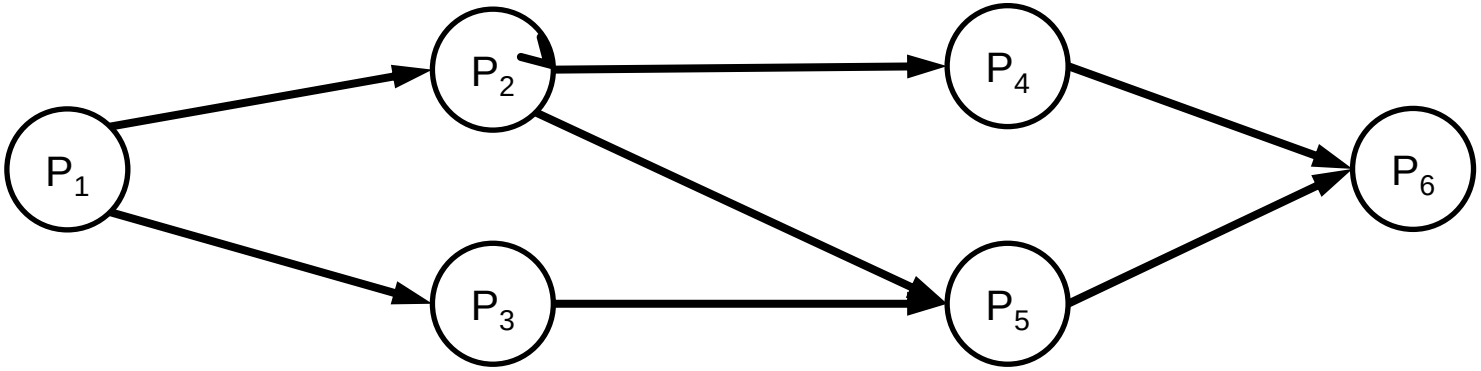
- Das vorangegangene Beispiel zeigt, wie ein einzelner Semaphor einen kritischen Abschnitt sichern kann.
- Das Anwendungsgebiet für Semaphore ist jedoch breiter. Mit Hilfe von Semaphoren lässt sich bspw.
  1. der Zugriff auf Ressourcen mit begrenzten Kapazitäten steuern (Erzeuger-Verbraucher-Problem),
  2. die Ausführungsreihenfolge nebenläufiger Prozesse festlegen.
- Beide Anwendungsfällte werden nun erläutert.

Ausführungsreihenfolge mittels Semaphoren

- Anwendungen bestehen oft aus **mehreren Prozessen**, von denen jeder eine **Teilaufgabe** übernimmt (z.B. Darstellung, Datenverarbeitung, Speichern im Hintergrund, etc.).
- Zwischen ihnen können **Datenabhängigkeiten** bestehen, d.h. ein Prozess kann erst dann einen bestimmten Arbeitsschritt erledigen, wenn ein anderer Prozess die dafür notwendigen Daten bereitgestellt hat (bspw. kann die Darstellung erst nach Datenverarbeitung erfolgen).
- Um *busy-waits* zu vermeiden, ist eine Implementierung dieser Abhängigkeiten mit Hilfe von Semaphoren sinnvoll.

Ausführungsreihenfolge mittels Semaphoren

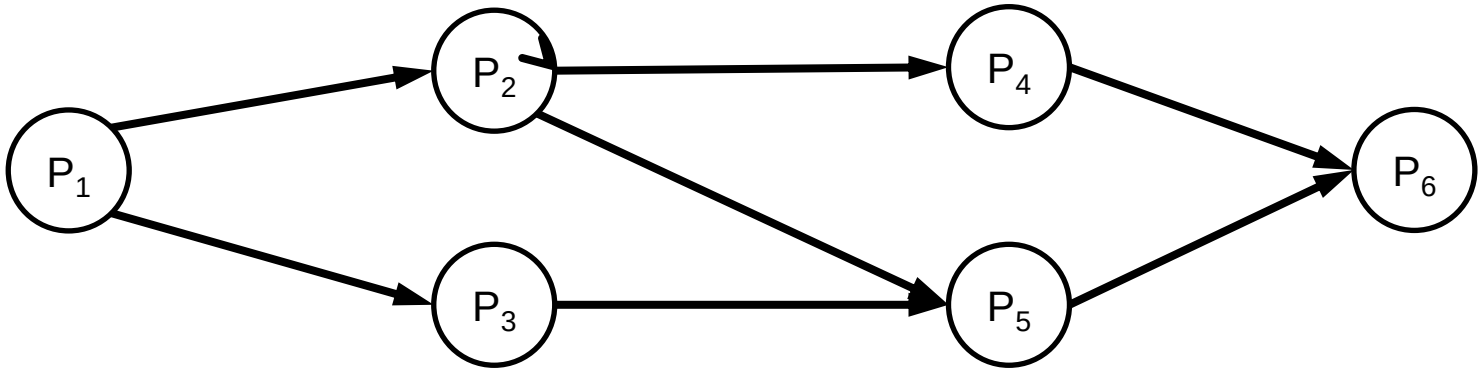
Die Datenabhängigkeiten können als **Präzedenzgraph** dargestellt werden, aus dem die resultierende Ausführungsreihenfolge der Prozesse intuitiv ablesbar ist:



Ausführungsreihenfolge mittels Semaphoren

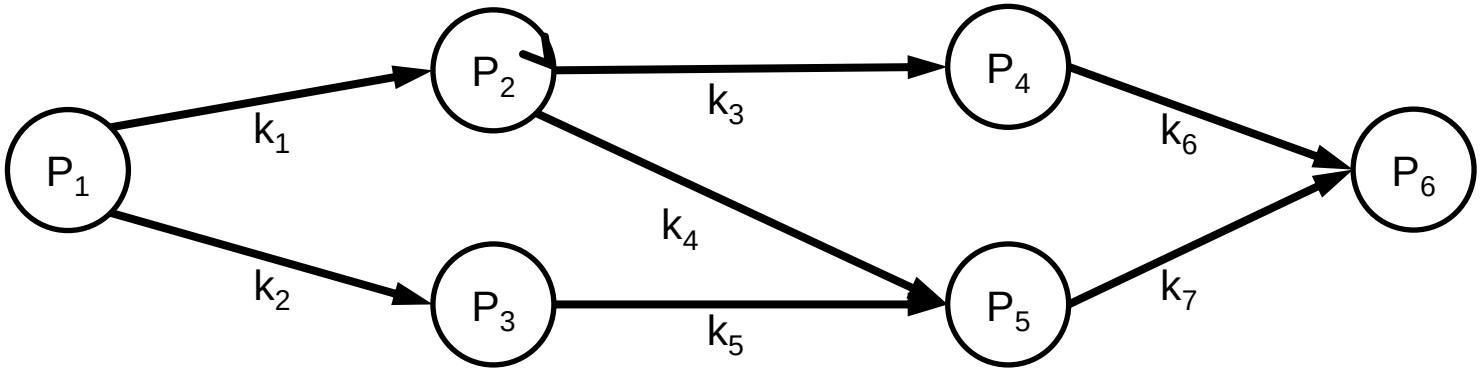
Implementierung mit Semaphoren:

- Für jede **Kante** im Präzedenzgraphen wird ein **Semaphor** angelegt und mit dem **Wert 0** initialisiert.
- Jeder an dem **Endpunkt** einer Kante liegende Prozess muss den der Kante zugeordneten Semaphor mit `P()` **passieren**.
- Der **Vorgängerprozess** erlaubt die Passage mit `V()`, sobald er seine eigene Arbeit beendet hat.



Ausführungsreihenfolge mittels Semaphoren

Beispiel:



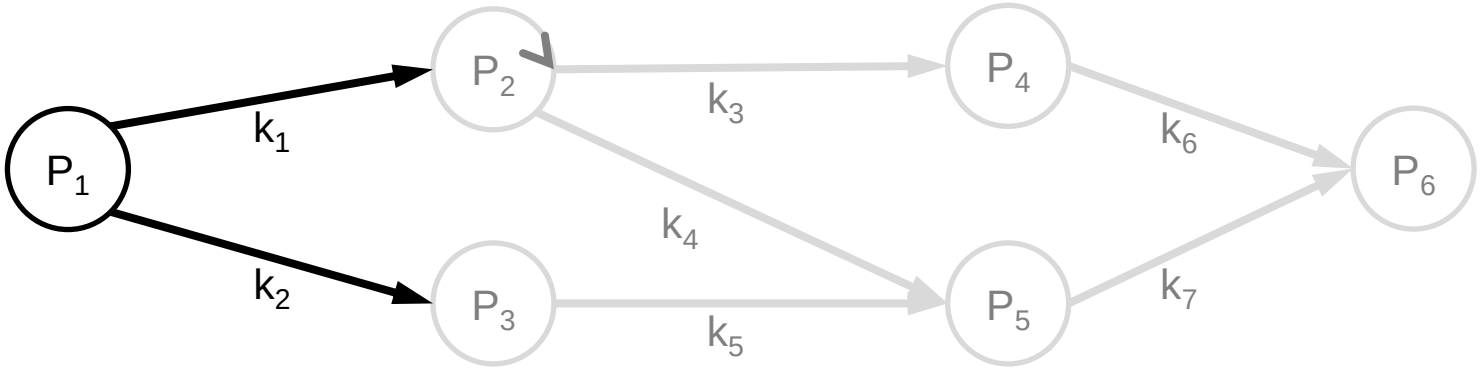
Initialisierung:

```
k1, k2, k3, k4, k5, k6, k7: Semaphor;
```

```
k1..k7.wert := 0;
```

Ausführungsreihenfolge mittels Semaphoren

Beispiel:

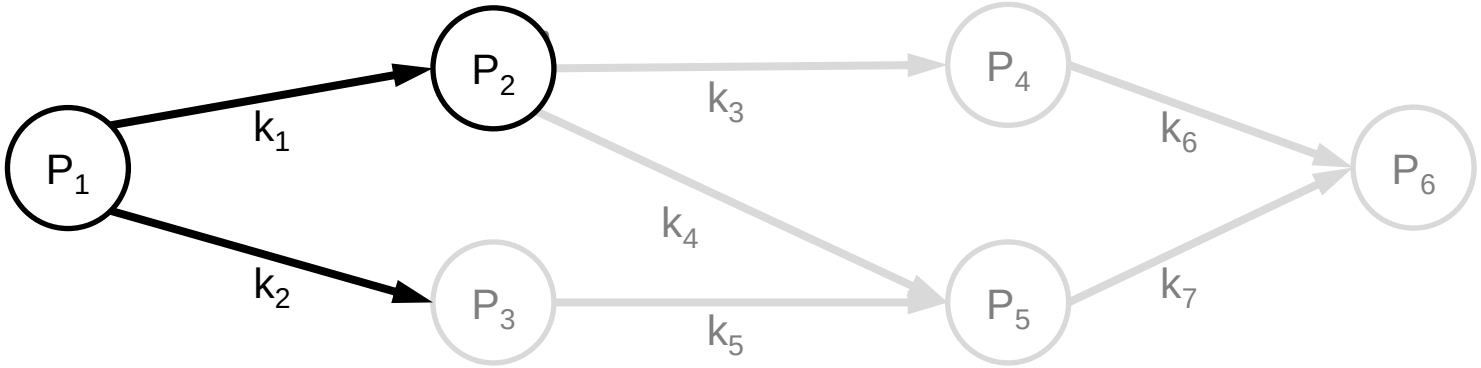


P1

```
...
V(k1); V(k2);
```

Ausführungsreihenfolge mittels Semaphoren

Beispiel:



P1

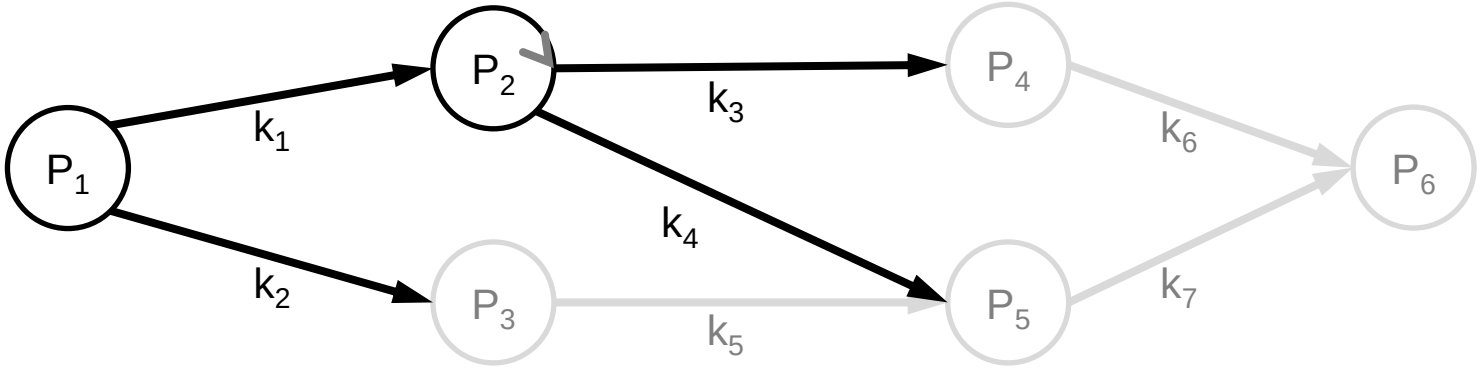
```
...
V(k1); V(k2);
```

P2

```
P(k1);
...
```

Ausführungsreihenfolge mittels Semaphoren

Beispiel:



P1

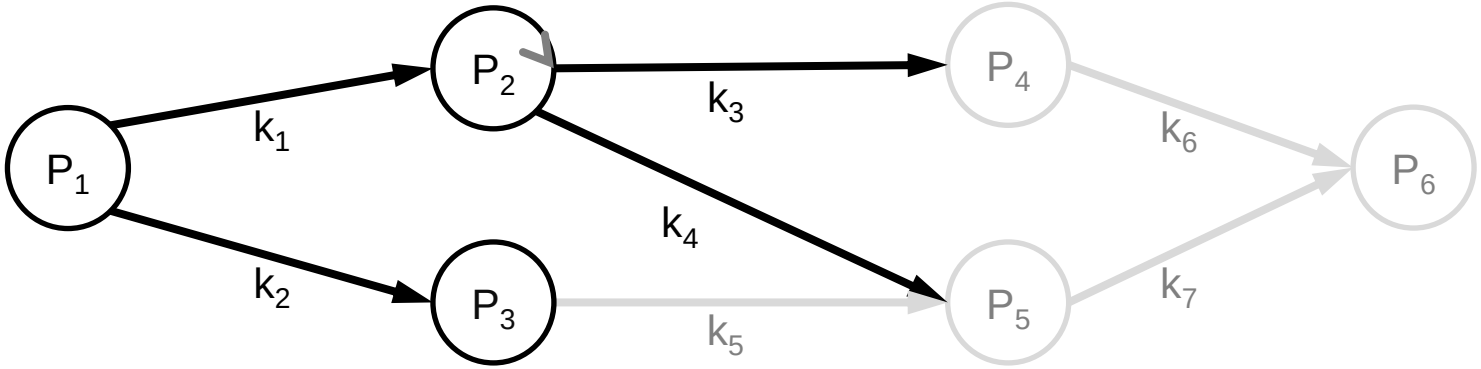
```
                    
...
V(k1); V(k2);
```

P2

```
P(k1);
...
V(k3); V(k4);
```

Ausführungsreihenfolge mittels Semaphoren

Beispiel:



P1

```
...
V(k1); V(k2);
```

P2

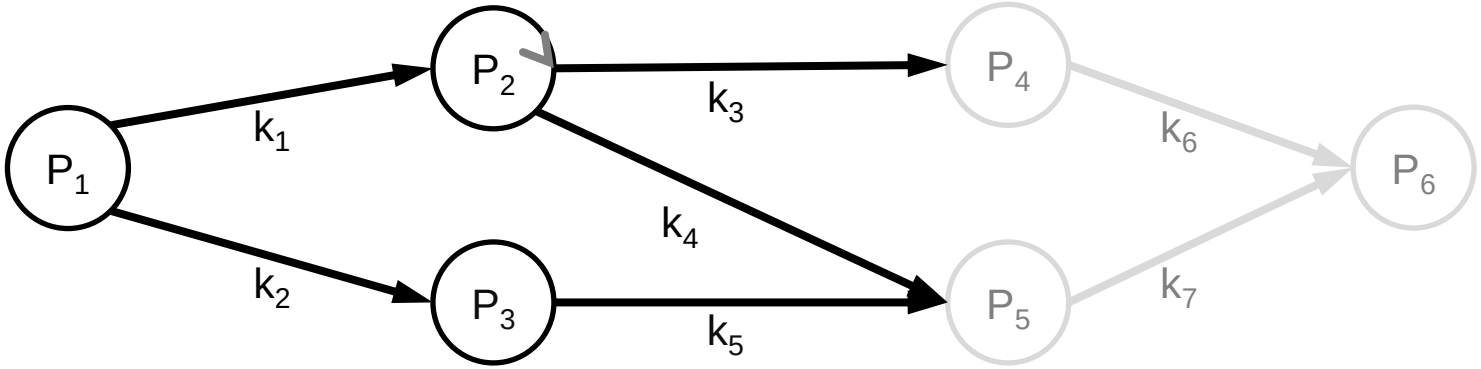
```
P(k1);
...
V(k3); V(k4);
```

P3

```
P(k2);
...
```

Ausführungsreihenfolge mittels Semaphoren

Beispiel:



P1

```
...
V(k1); V(k2);
```

P2

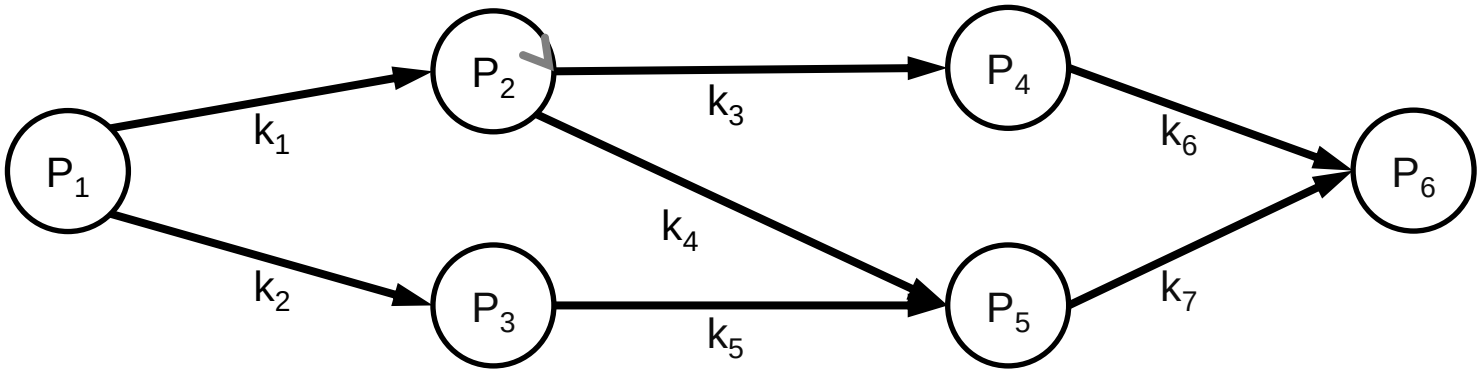
```
P(k1);
...
V(k3); V(k4);
```

P3

```
P(k2);
...
V(k5);
```

Ausführungsreihenfolge mittels Semaphoren

Beispiel:



P1

```
...
V(k1); V(k2);
```

P2

```
P(k1);
...
V(k3); V(k4);
```

P3

```
P(k2);
...
V(k5);
```

P4

```
P(k3);
...
V(k6);
```

P5

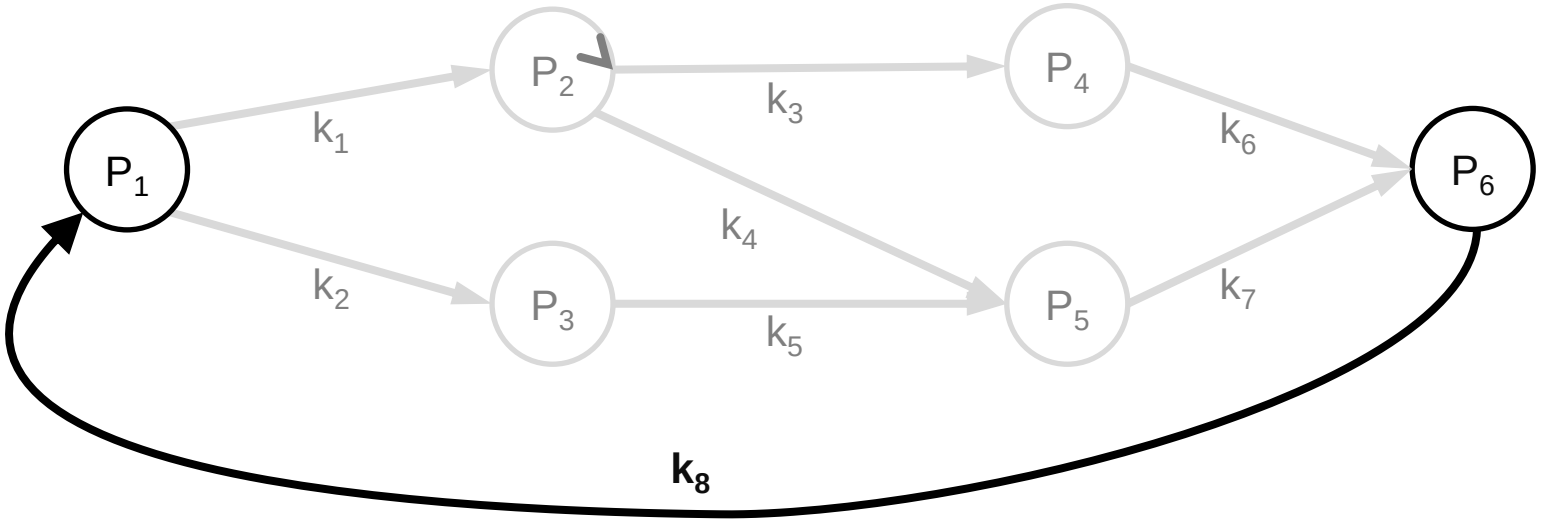
```
P(k4); P(k5);
...
V(k7);
```

P6

```
P(k6); P(k7)
...
```

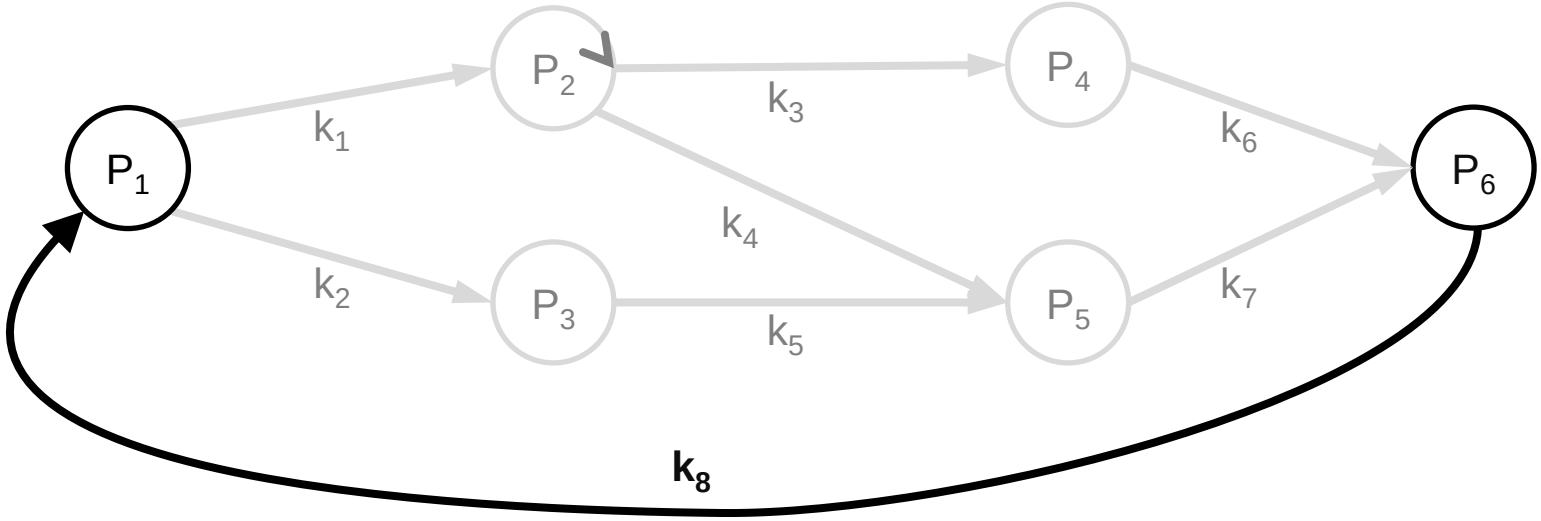
Ausführungsreihenfolge mittels Semaphoren

Bei **zyklischen** Prozessen muss auch der **Startprozess** mindestens*) einen Semaphor passieren. Dessen Wert wird zu Beginn **mit 1** initialisiert, damit der Prozess als erster starten kann.



*) Gibt es mehrere Endknoten im Präzedenzgraphen, so führt von **jedem** dieser Knoten eine Kante zurück zum Startknoten.

Ausführungsreihenfolge mittels Semaphoren



Initialisierung:

```
k1, k2, k3, k4, k5, k6, k7, k8: Semaphor;
k1..k7.wert := 0; k8.wert := 1;
```

P1

```
P(k8);
...
V(k1); V(k2);
```

...

P6

```
P(k6); P(k7)
...
V(k8);
```

Zusammenfassung

- Semaphoren sind **vom Betriebssystem bereitgestellte** Mittel zur Prozesssynchronisation,
- ***busy-waits*** in Userprozessen können (nahezu) vollständig vermieden werden,
- **Fehleranfälligkeit** aufgrund der Standardisierung wesentlich geringer als bei individueller Implementierung durch Anwendungsentwickler,
- **einige**[*)] **Anwendungsfälle**:
  - gegenseitiger Ausschluss,
  - Datenbereitstellung und –verbrauch
  - Ausführungsreihenfolge bei Datenabhängigkeiten

[*)] Wir kommen später noch auf zwei weitere Anwendungsfälle zurück