



Prozesssynchronisation I

Nebenläufigkeit

Moderne Rechnerarchitekturen erlauben es typischerweise, mehrere Prozesse (oder Threads) **gleichzeitig** auszuführen. Die jeweils laufenden Prozesse werden als **nebenläufige** oder **konkurrente Prozesse** bezeichnet.

Selbst bei Einkernprozessoren können Prozesse bei Vorhandensein von Unterbrechungen^{*)} als nebenläufig betrachtet werden. Nebenläufigkeit bezieht sich also auf **alle** zur Zeit im System **existierenden lauffähigen** Prozesse.

Problemstellung: Dort, wo nebenläufige Prozesse auf dieselben Daten oder Betriebsmittel zugreifen können, entsteht ggf. Notwendigkeit zur **Synchronisation**.

^{*)} Dazu zählt auch das Round-Robin-Scheduling!

Deadlock Reihenfolgeplanung
Semaphoren
kritische Abschnitte
Ressourcenallokation
dinierende Philosophen

Betriebsmittel

Als **Betriebsmittel** versteht man all diejenigen **Systemelemente**, die ein Prozess zu seiner **vollständigen** und **korrekten Ausführung** benötigt und die ihm über das Betriebssystem bereitgestellt werden. Dazu gehören unter anderem:

- Geräte (Drucker, Zeigegeräte, Tastatur)
- Speicher (Hauptspeicher, Sekundärspeicher)
- Kommunikationskanäle (Netzwerk)
- etc.

Kritische Abschnitte

Bei mindestens zwei nebenläufigen Prozessen, die gemeinsame Daten oder Betriebsmittel nutzen können, unterscheidet man:

1. **unkritische Abschnitte**, in denen **keine gemeinsamen Daten oder Betriebsmittel** genutzt werden oder auf die von allen beteiligten Prozessen *nur lesend* zugegriffen wird,
2. **kritische Abschnitte**, in denen von mindestens einem Prozess *schreibend* auf gemeinsam genutzte Daten oder Betriebsmittel zugegriffen wird

Ein Abschnitt ist stets **entweder** kritisch **oder** unkritisch, er kann nie beides zugleich sein.

Kritische Abschnitte – Beispiel

Warteschlange eines Druckers:

	Position	Dokument
	1	anschreiben.docx
	2	anlage1.pdf
<code>frei</code> →	3	
	4	
	5	

Variable `frei` zeigt auf nächsten freien Platz in der Warteschlange.

Kritische Abschnitte – Beispiel

	Position	Dokument
	1	anschreiben.docx
	2	anlage1.pdf
frei →	3	
	4	
	5	

Prozess A:

```
p := frei  
einfuegen("anlage2.pdf", p)  
frei := p + 1
```

Prozess B:

```
p := frei  
einfuegen("bild1.jpg", p)  
frei := p + 1
```

Kritische Abschnitte – Erwarteter Ausgang

Position	Dokument
1	anschreiben.docx
2	anlage1.pdf
3	anlage2.pdf
4	bild1.jpg
frei → 5	

Prozess A:

```
p := frei  
einfuegen("anlage2.pdf", p)  
frei := p + 1
```

Prozess B:

```
p := frei  
einfuegen("bild1.jpg", p)  
frei := p + 1
```


Kritische Abschnitte – Möglicher Ablauf mit Unterbrechung

Position	Dokument
1	anschreiben.docx
2	anlage1.pdf
3	
4	
5	

p → frei →

3

4

5

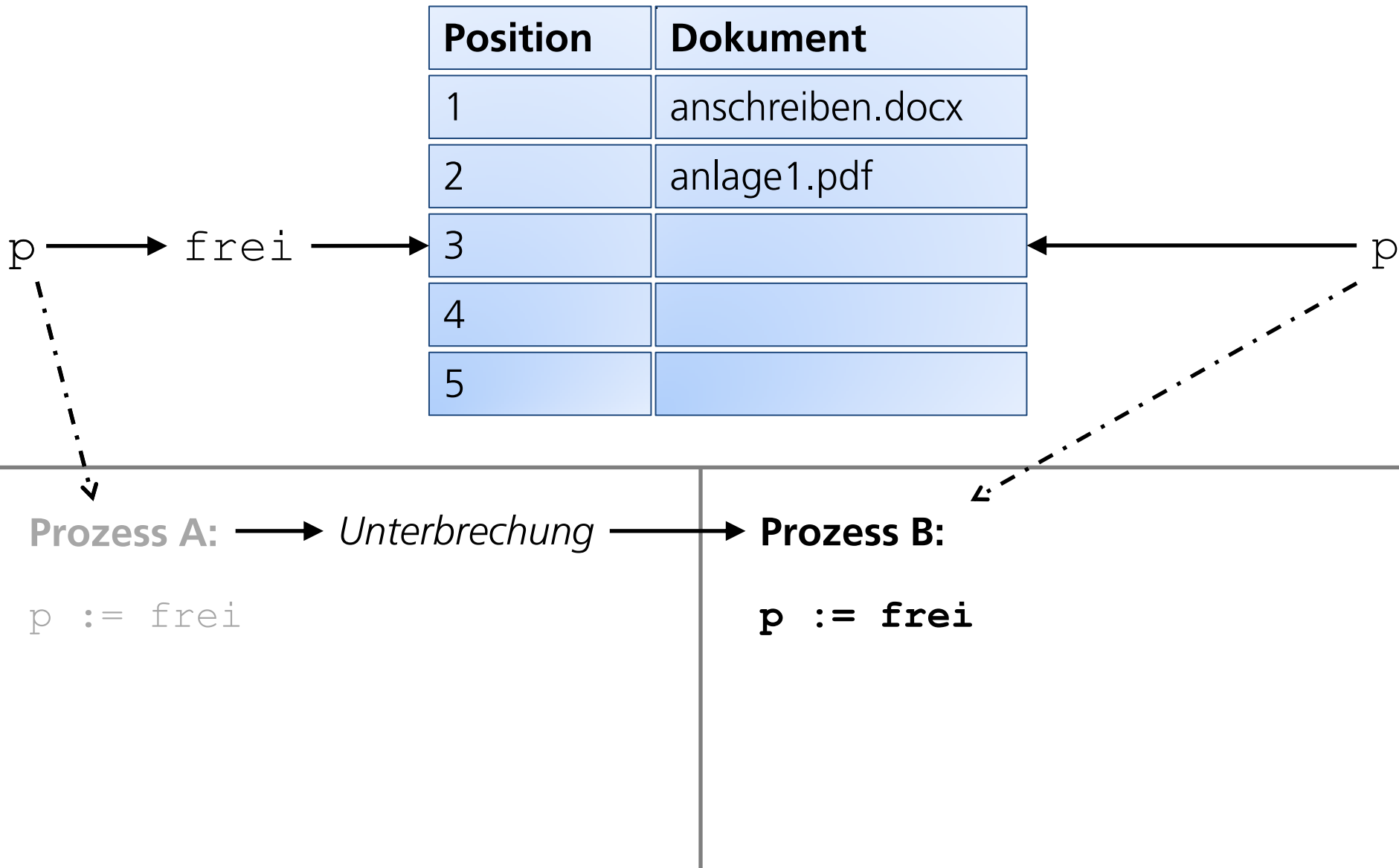
↓

Prozess A:

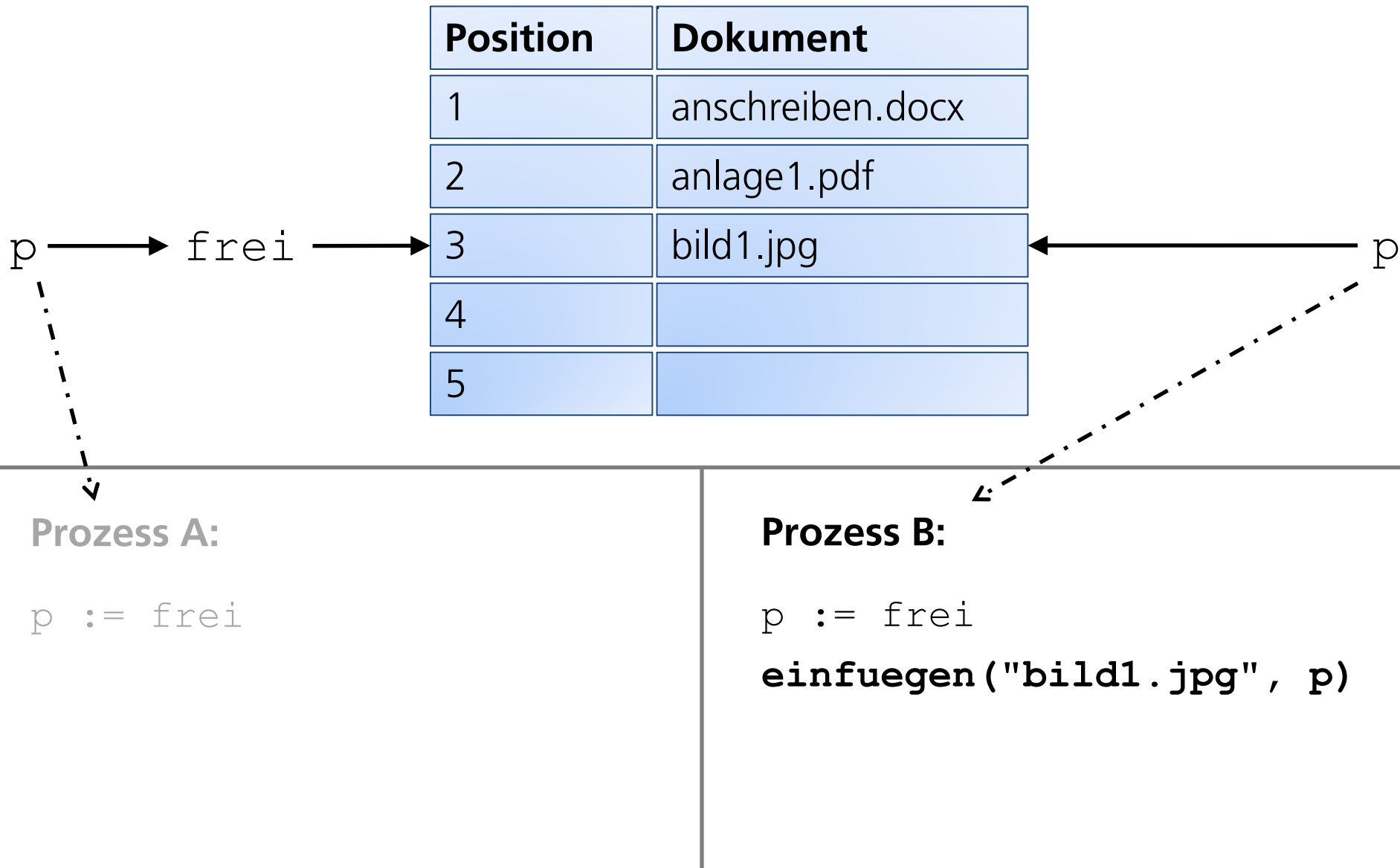
p := frei

Prozess B:

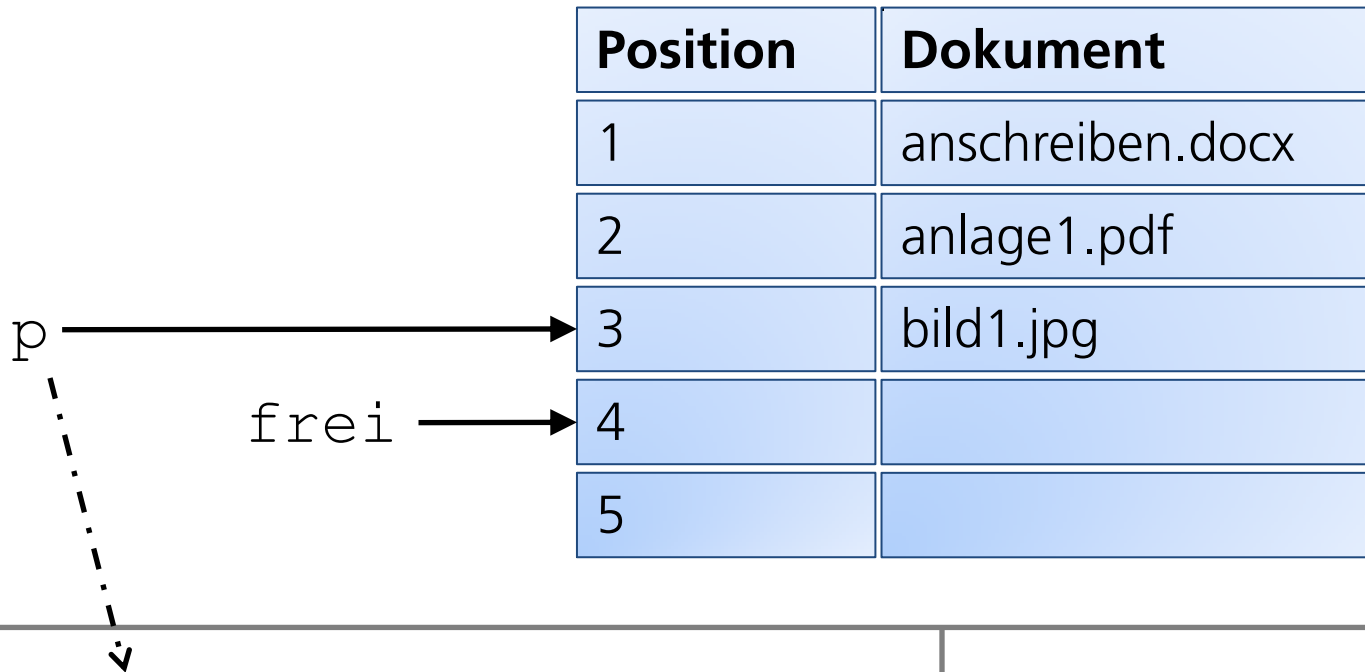
Kritische Abschnitte – Möglicher Ablauf mit Unterbrechung



Kritische Abschnitte – Möglicher Ablauf mit Unterbrechung



Kritische Abschnitte – Möglicher Ablauf mit Unterbrechung



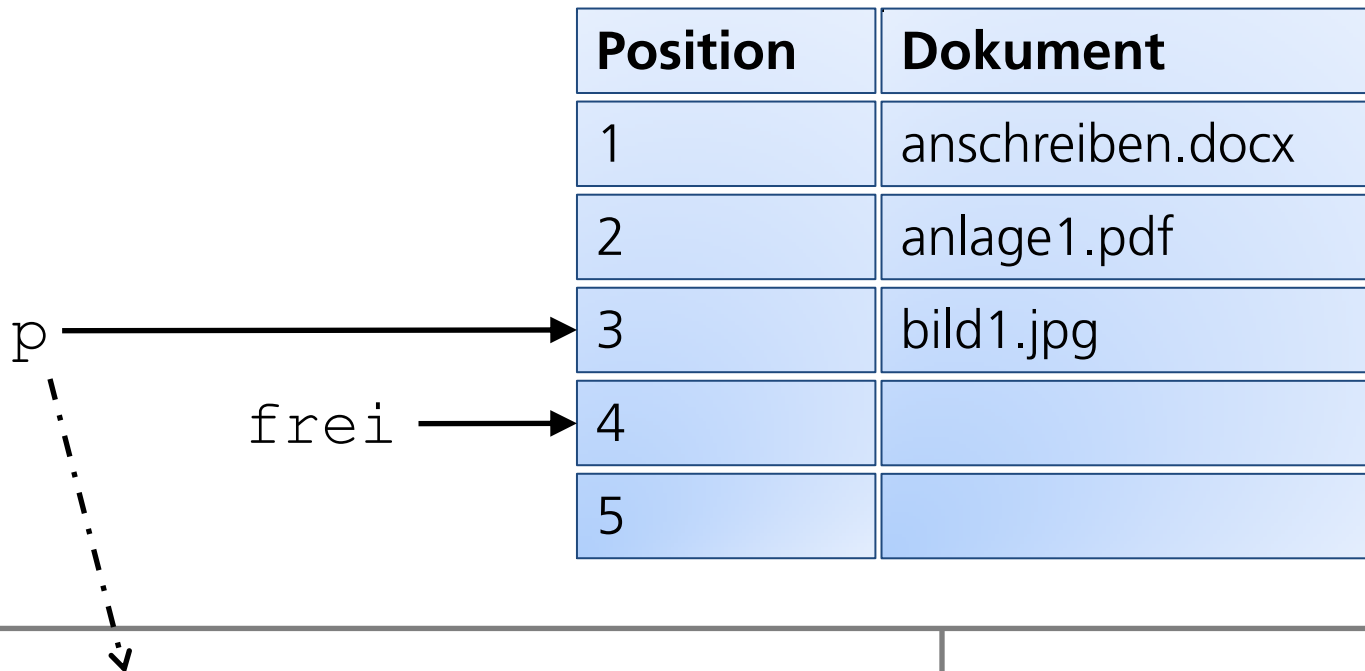
Prozess A:

```
p := frei
```

Prozess B:

```
p := frei  
einfuegen("bild1.jpg", p)  
frei := p + 1
```

Kritische Abschnitte – Möglicher Ablauf mit Unterbrechung



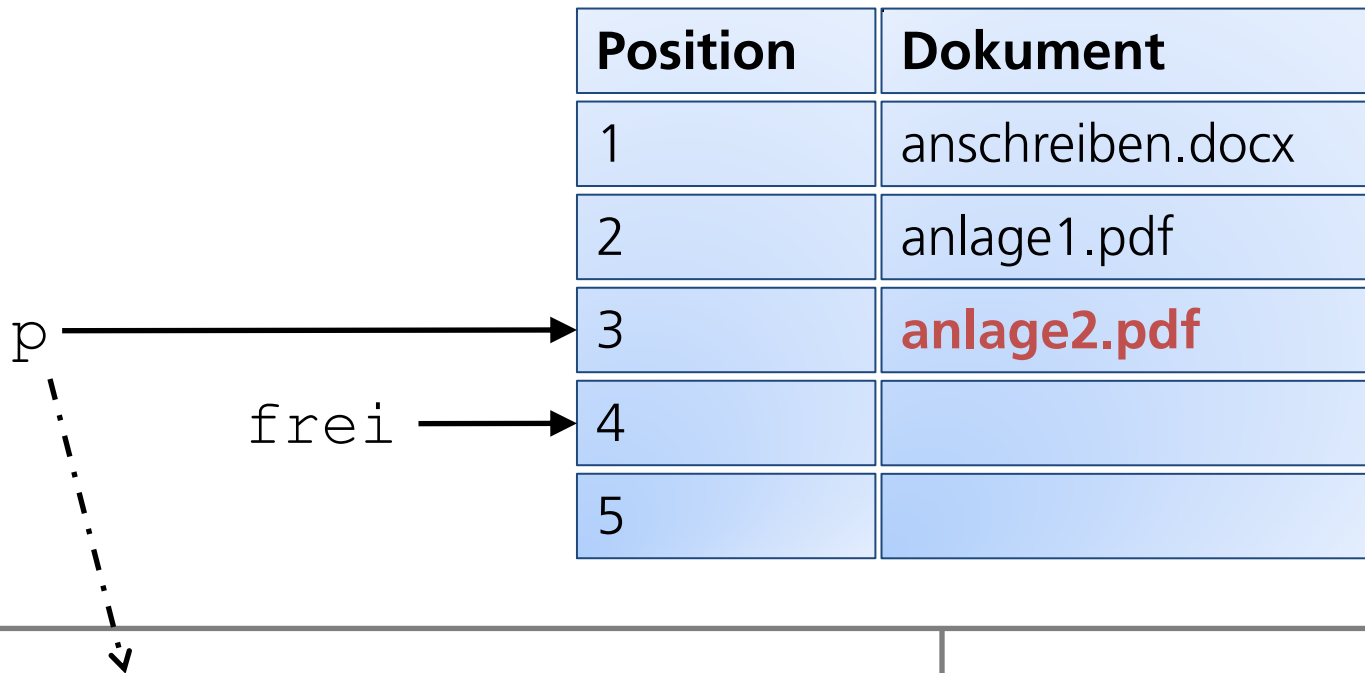
Prozess A: ← zurück zu A ←

$p := \text{frei}$

Prozess B:

```
p := frei  
einfuegen("bild1.jpg", p)  
frei := p + 1
```

Kritische Abschnitte – Möglicher Ablauf mit Unterbrechung



Prozess A:

`p := frei`

`einfuegen("anlage2.pdf", p)`

Prozess B:

`p := frei`

`einfuegen("bild1.jpg", p)`

`frei := p + 1`

Kritische Abschnitte – Möglicher Ablauf mit Unterbrechung

Position	Dokument
1	anschreiben.docx
2	anlage1.pdf
3	anlage2.pdf
frei → 4	
5	

Prozess A:

```
p := frei  
einfuegen("anlage2.pdf", p)  
frei := p + 1
```

Prozess B:

```
p := frei  
einfuegen("bild1.jpg", p)  
frei := p + 1
```

Kritische Abschnitte – Möglicher Ablauf mit Unterbrechung

	Position	Dokument
	1	anschreiben.docx
	2	anlage1.pdf
	3	anlage2.pdf
frei →	4	
	5	

Resultat: Die Warteschlange ist wieder konsistent, aber ein Auftrag ist verlorengegangen!

→ **Synchronisation** ist notwendig

Race Conditions

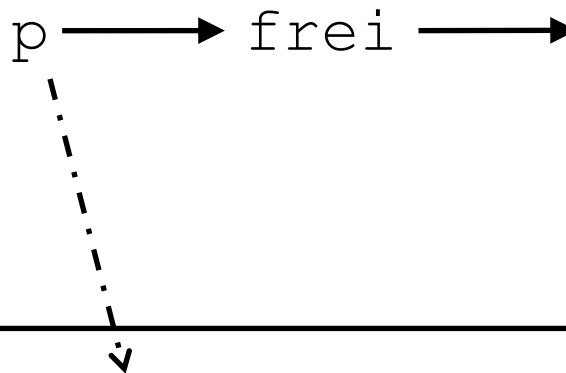
Da der Zeitpunkt von Unterbrechungen **nicht vorhersagbar** ist, können unterschiedliche, schwer zu kontrollierende Effekte eintreten, die davon abhängen, welcher Prozess zuerst bestimmte Aktionen ausführen kann. Diese Abhängigkeiten nennt man **Race-Conditions** („Wettrennen“).

Im vorangegangenen Beispiel:

- Unterbrechung **vor oder nach** Einfügen/Zähler verändern?
 - Geht ein Dokument verloren?
 - Verliert A oder B sein Dokument?
- Welcher Prozess **startet** zuerst?
 - Verliert A oder B sein Dokument?

Kritische Abschnitte – Alternativer Ablauf

Position	Dokument
1	anschreiben.docx
2	anlage1.pdf
3	anlage2.pdf
4	
5	

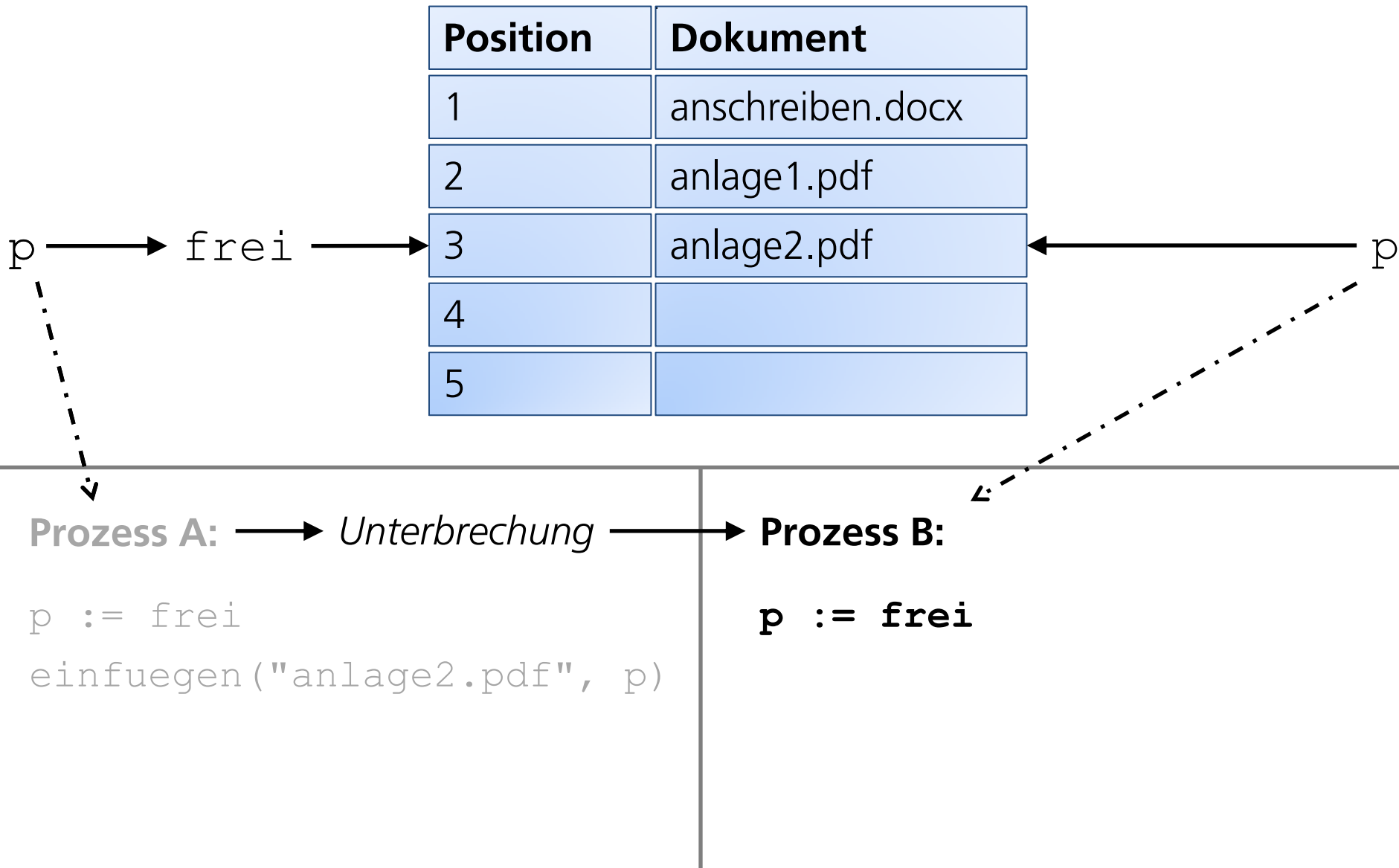


Prozess A:

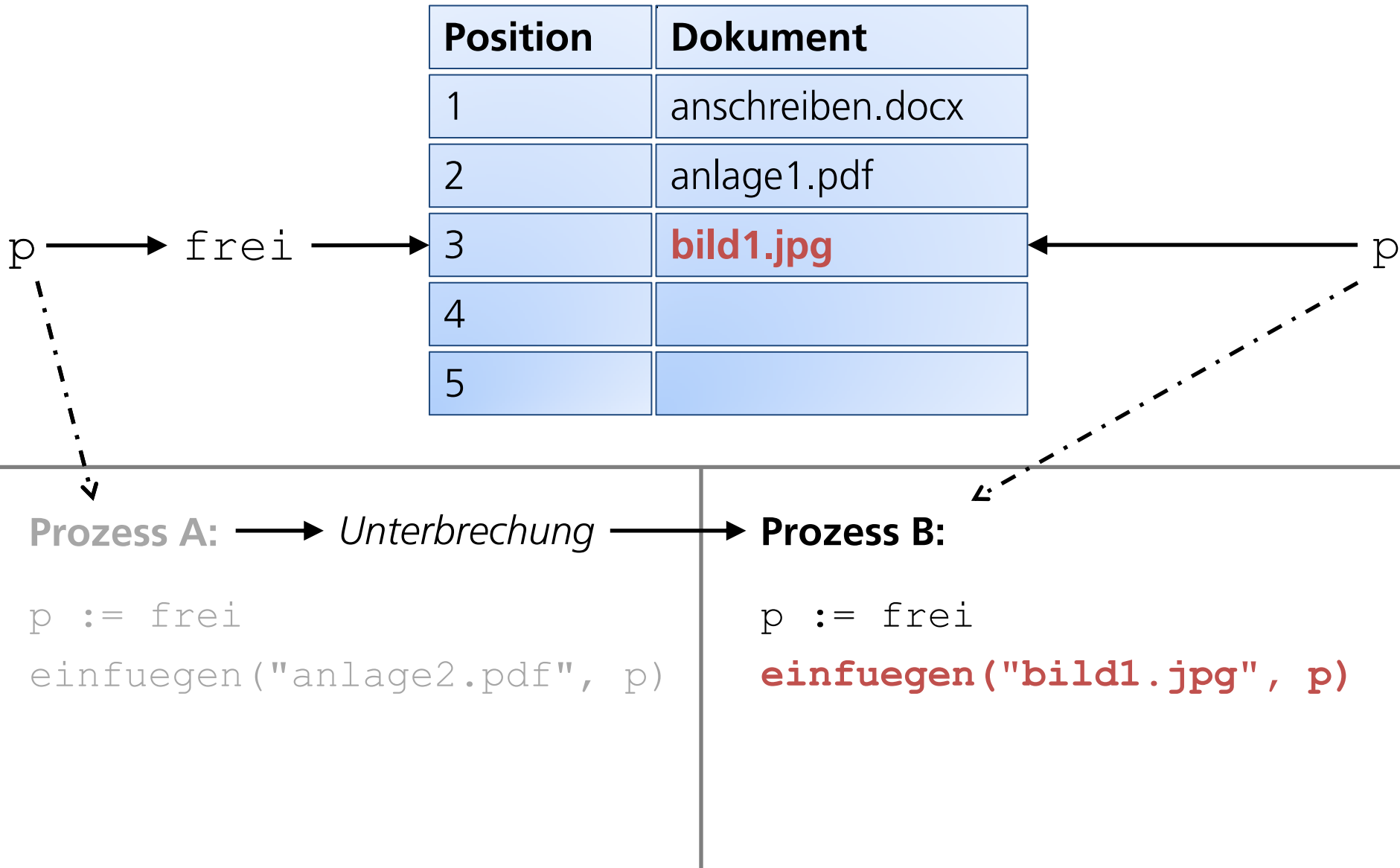
```
p := frei  
einfuegen("anlage2.pdf", p)
```

Prozess B:

Kritische Abschnitte – Alternativer Ablauf



Kritische Abschnitte – Alternativer Ablauf



Kritische Abschnitte – Alternativer Ablauf – Resultat

	Position	Dokument
	1	anschreiben.docx
	2	anlage1.pdf
	3	bild1.jpg
frei →	4	
	5	

Resultat: Wieder ist die Warteschlange am Ende konsistent, trotzdem ist ebenfalls ein Auftrag verlorengegangen – aber diesmal der andere!

→ **Race-Condition**, Synchronisation ist notwendig.

(An-)forderungen an kritische Abschnitte (Dijkstra 1965)

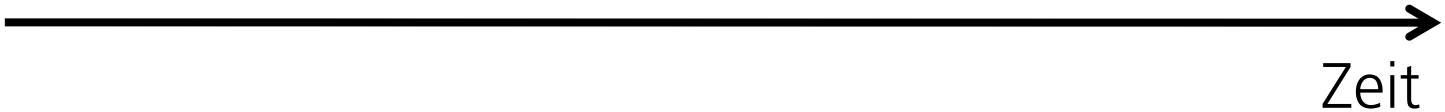
1. Zwei Prozesse dürfen **nicht gleichzeitig** in demselben kritischen Abschnitt sein (*mutual exclusion*, kurz: *mutex*).
2. Jeder Prozess, der am Eingang eines kritischen Abschnitts wartet, muß irgendwann diesen Abschnitt auch betreten dürfen. **Kein ewiges Warten** darf möglich sein (*fairness condition*).
3. Ein Prozess darf außerhalb eines kritischen Abschnitts einen anderen Prozess **nicht blockieren**.
4. Es dürfen **keine Annahmen** über die Abarbeitungsgeschwindigkeit oder die Anzahl von Prozessen bzw. Prozessoren gemacht werden.

Gegenseitiger Ausschluss

Prozess A

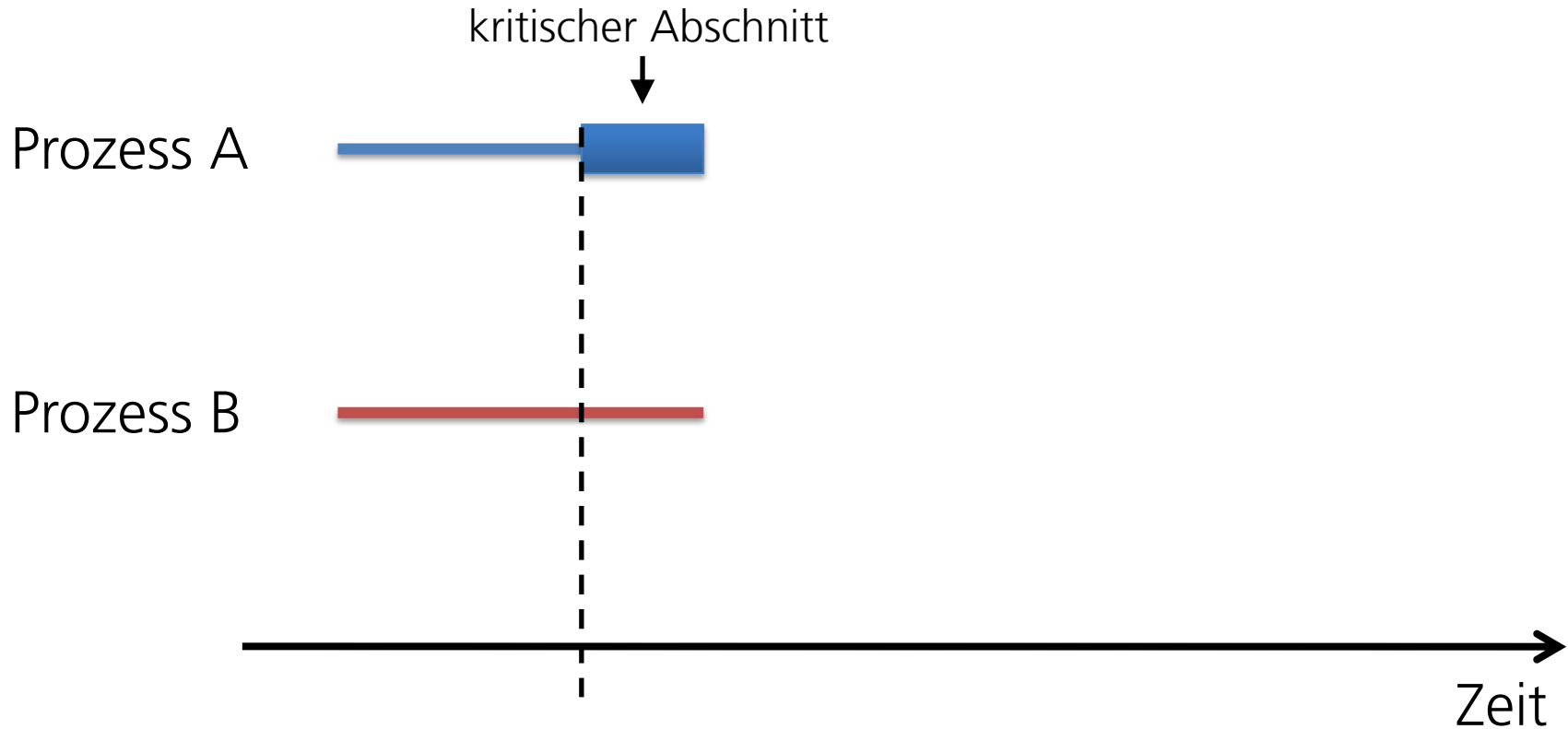


Prozess B



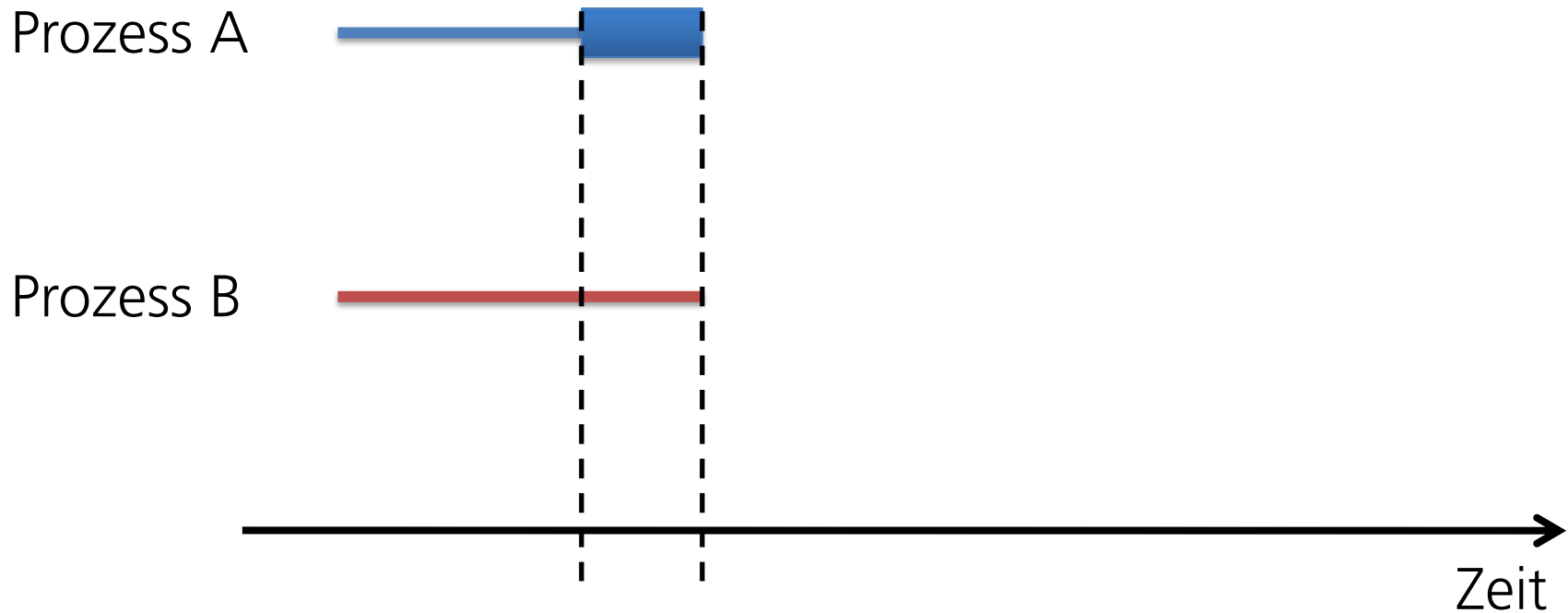
1. Beide Prozesse sind in unkritischen Abschnitten

Gegenseitiger Ausschluss



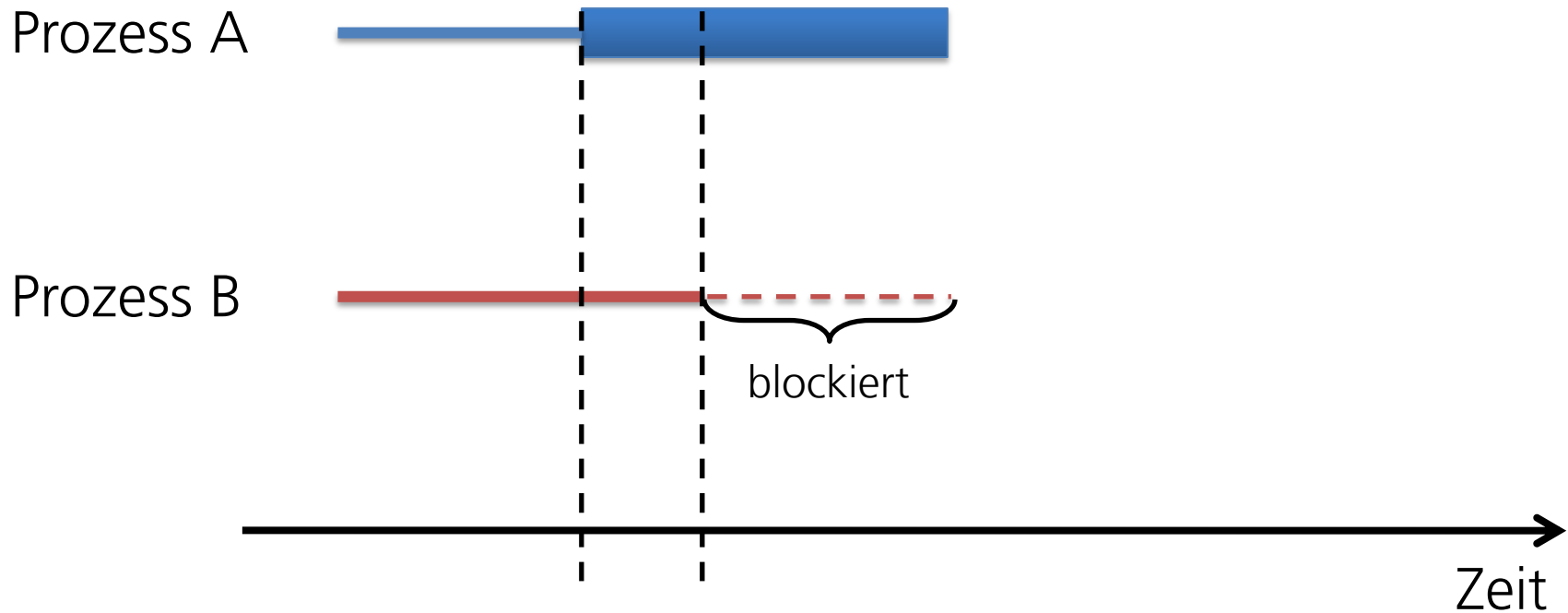
2. Prozess A betritt kritischen Abschnitt
Prozess B verbleibt in unkritischem Abschnitt

Gegenseitiger Ausschluss



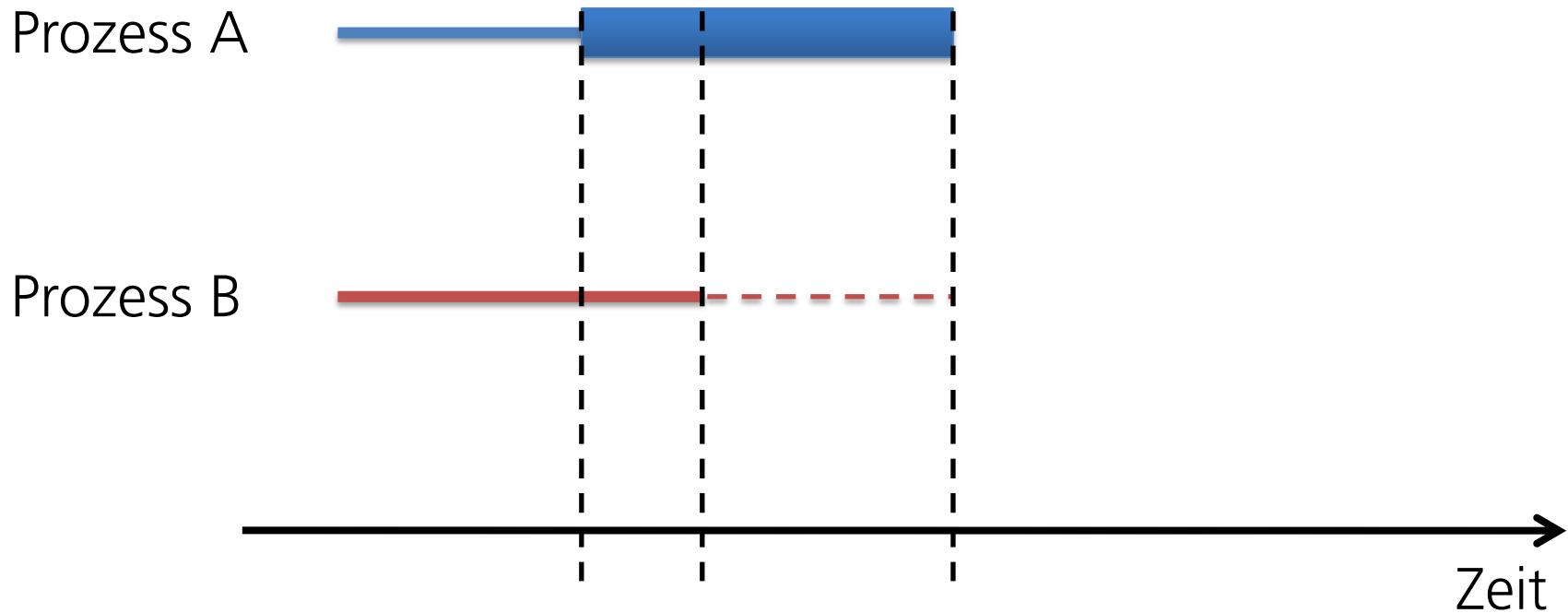
3. Prozess B versucht ebenfalls, kritischen Abschnitt zu betreten, den A noch nicht verlassen hat.

Gegenseitiger Ausschluss



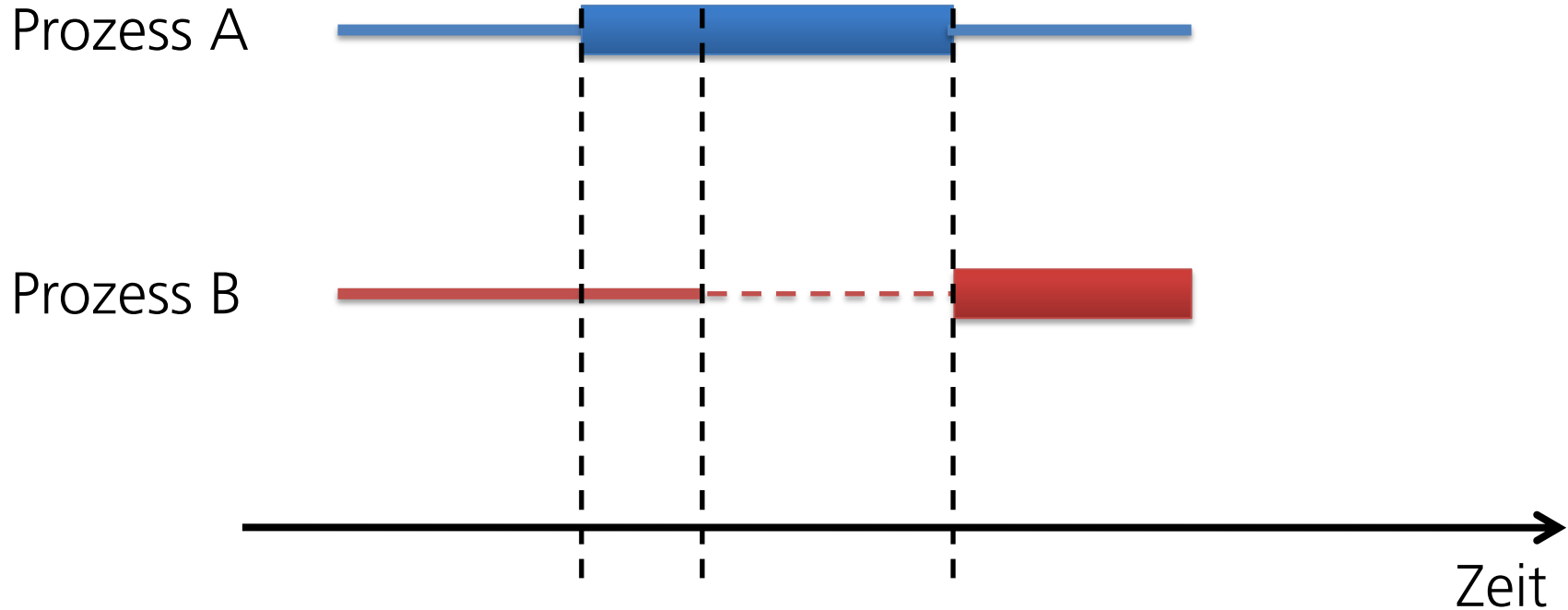
4. Prozess B wechselt in den Zustand „blockiert“

Gegenseitiger Ausschluss



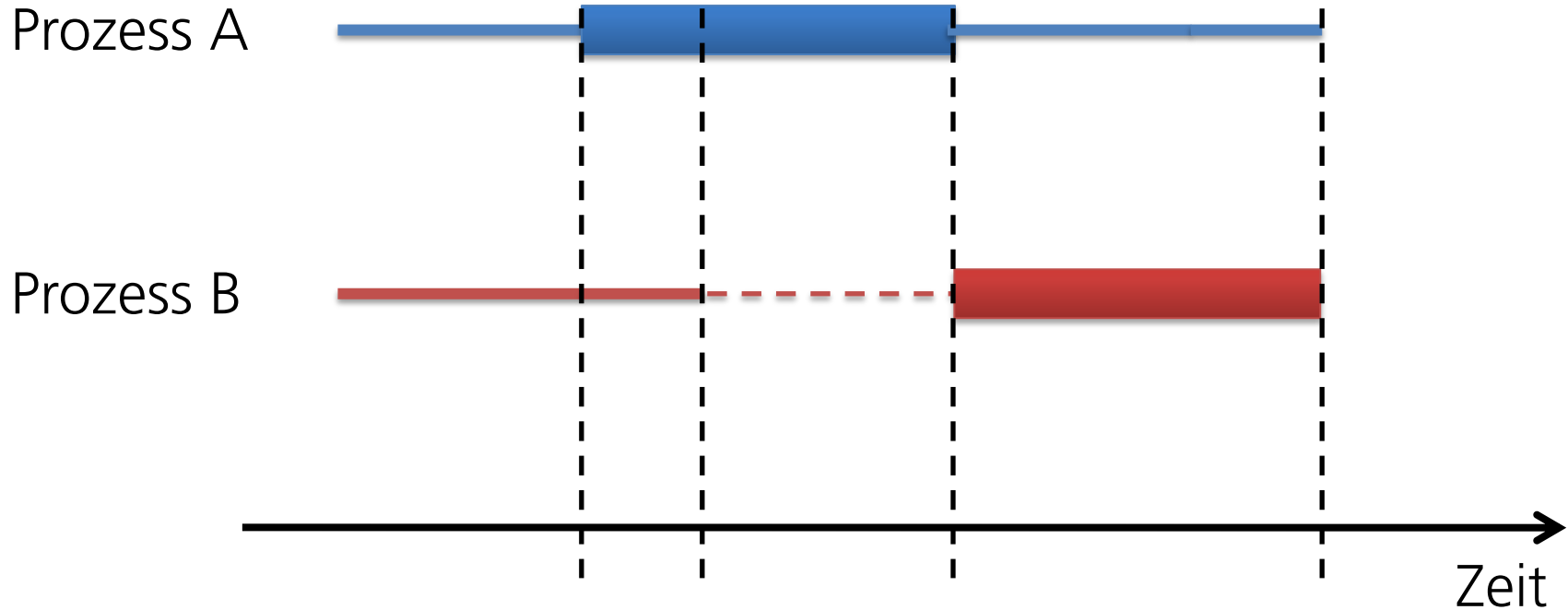
5. Prozess A beendet kritischen Abschnitt
Prozess B wird in den Zustand „bereit“ versetzt

Gegenseitiger Ausschluss



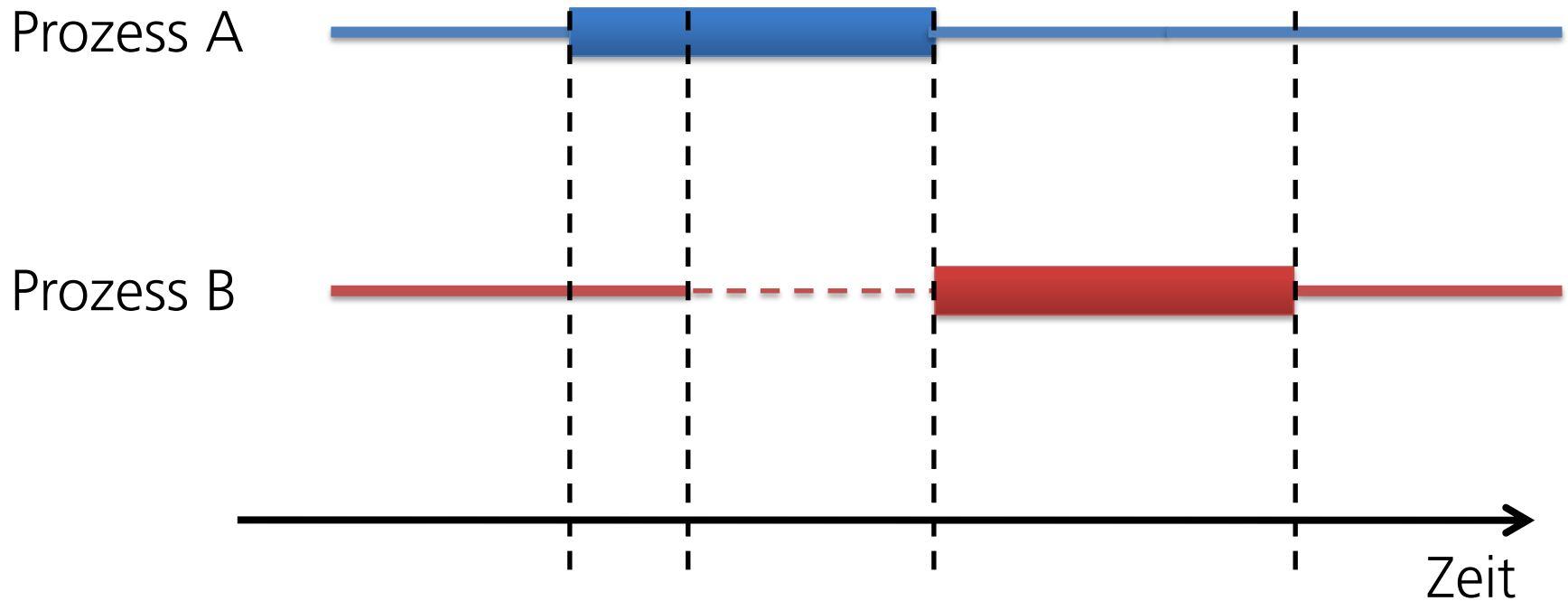
6. Prozess B kann kritischen Abschnitt betreten, Prozess A ist in unkritischem Abschnitt und kann daher weiterrechnen

Gegenseitiger Ausschluss



7. Prozess B beendet kritischen Abschnitt

Gegenseitiger Ausschluss



8. Beide Prozesse rechnen in unkritischen Abschnitten weiter

Verfahren zum gegenseitigen Ausschluss

Vorbemerkung: Wir betrachten zur Erläuterung der Verfahren hier **zyklische Prozesse** mit jeweils einem kritischen Abschnitt, die nach folgendem Schema abgearbeitet werden:

wiederhole

unkritischer Abschnitt

Eintritt in kritischen Abschnitt

kritischer Abschnitt

Austritt aus kritischem Abschnitt

unkritischer Abschnitt

bis Abbruch

Die im folgenden beschriebenen Szenarien gelten auch für azyklische Prozesse oder mehrere kritische Abschnitte.

Verfahren zum gegenseitigen Ausschluss

Grundsätzlich kommt eine Vielzahl Verfahren zur Sicherstellung gegenseitigen Ausschlusses infrage:

- **falloptimierte** Softwarelösungen für zwei oder mehr Prozesse
- vom Betriebssystem bereitgestellte **generische** Lösungen

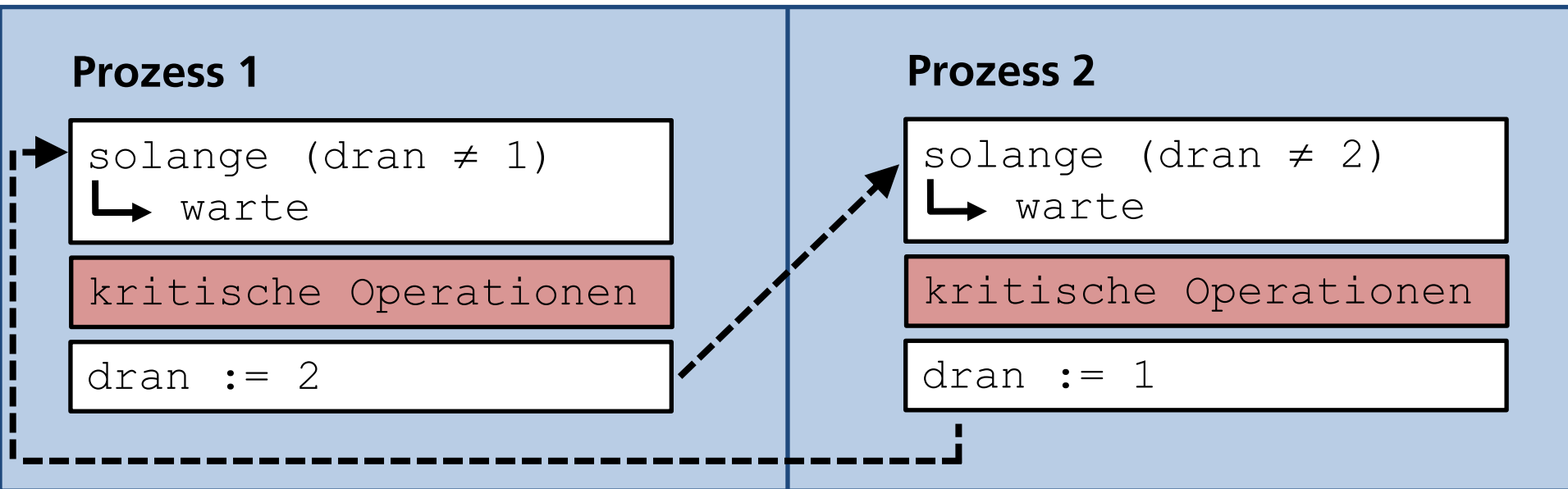
Für diese beiden Varianten ist **Hardwareunterstützung** durch spezielle CPU-Operationen sehr hilfreich.

Außerdem sind noch interessant:

- nachrichtenbasierte Lösungen
- dateisystemspezifische Lösungen (nicht Teil des Kurstextes)

Softwarelösung – Erster Ansatz

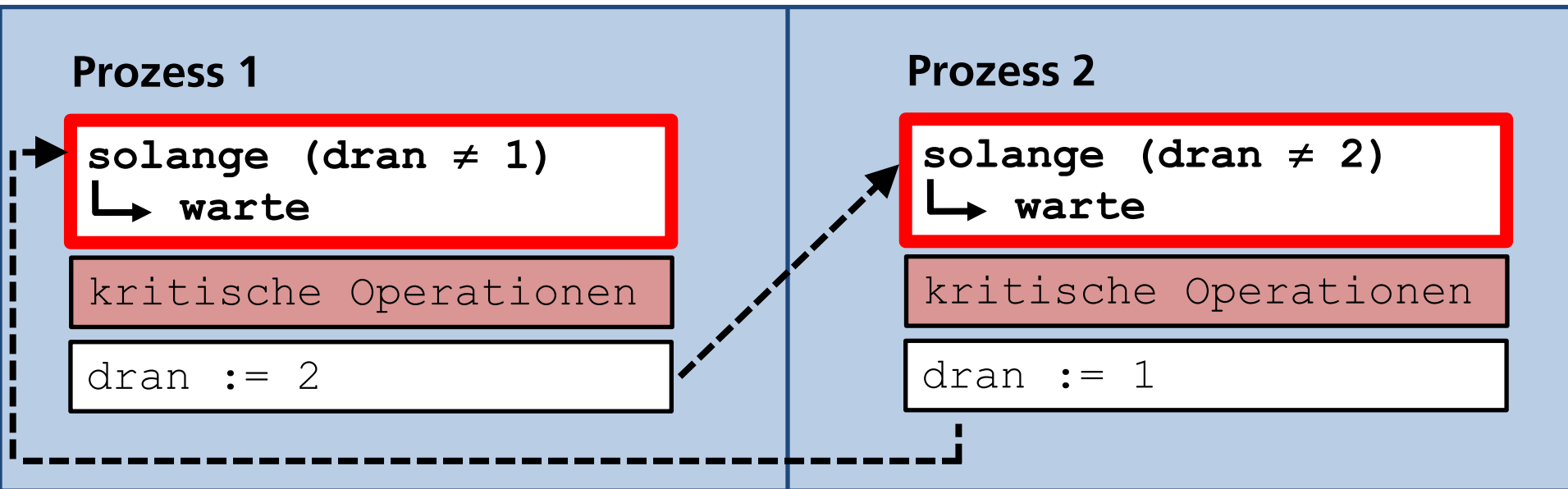
Zutrittsregelung mittels globaler **Zugangsvariablen**, deren Wert festlegt, welcher Prozess als nächstes den kritischen Abschnitt betreten darf:



Hinweis: `dran` wird mit 1 oder 2 initialisiert

Softwarelösung – Erster Ansatz

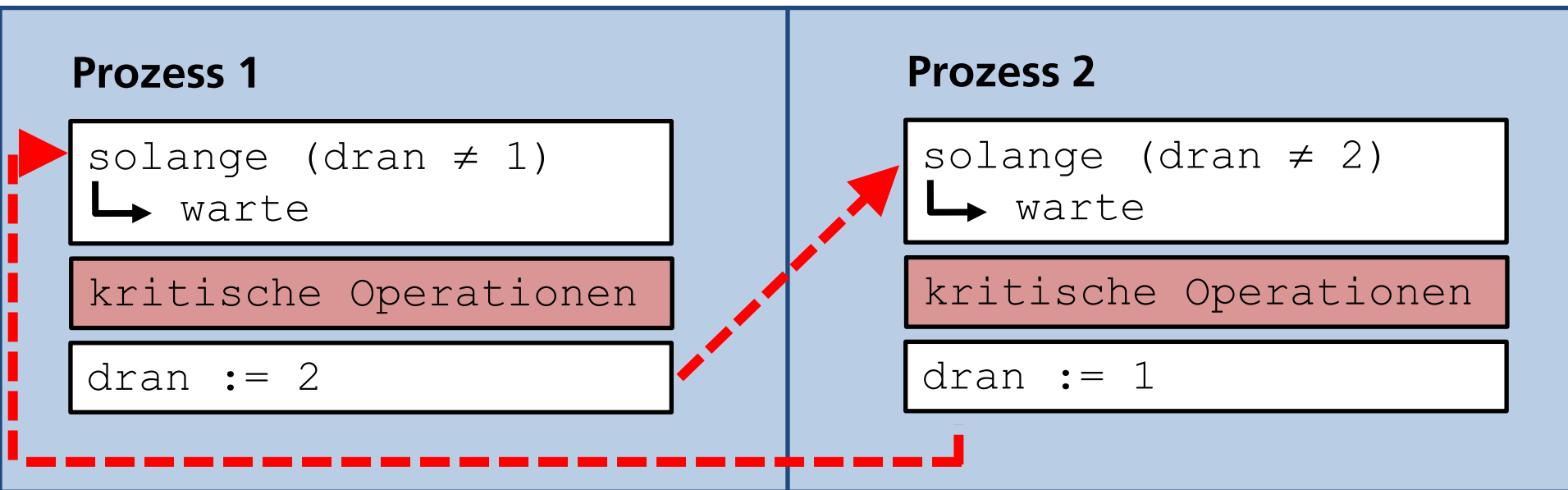
Zutrittsregelung mittels globaler **Zugangsvariablen**, deren Wert festlegt, welcher Prozess als nächstes den kritischen Abschnitt betreten darf:



Probleme: 1. beschäftigtes Warten (*busy wait*)

Softwarelösung – Erster Ansatz

Zutrittsregelung mittels globaler **Zugangsvariablen**, deren Wert festlegt, welcher Prozess als nächstes den kritischen Abschnitt betreten darf:

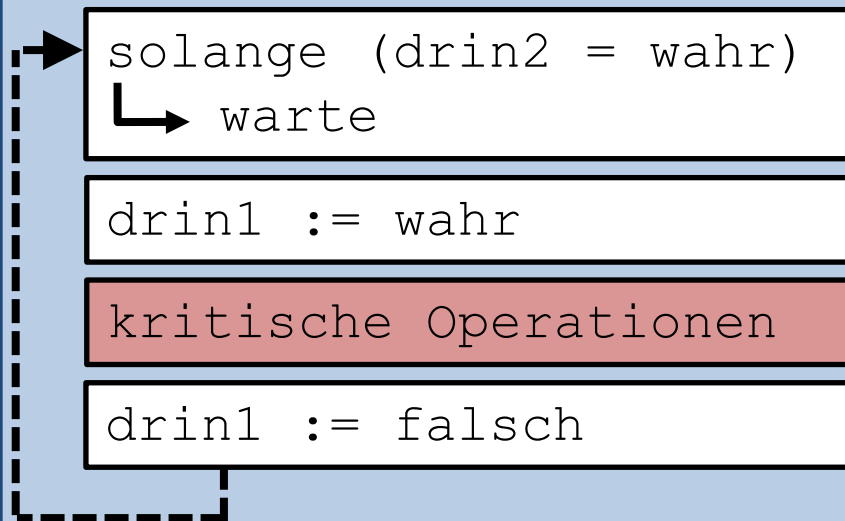


Probleme: 1. beschäftigtes Warten (*busy wait*)
2. erzwungene Abwechslung

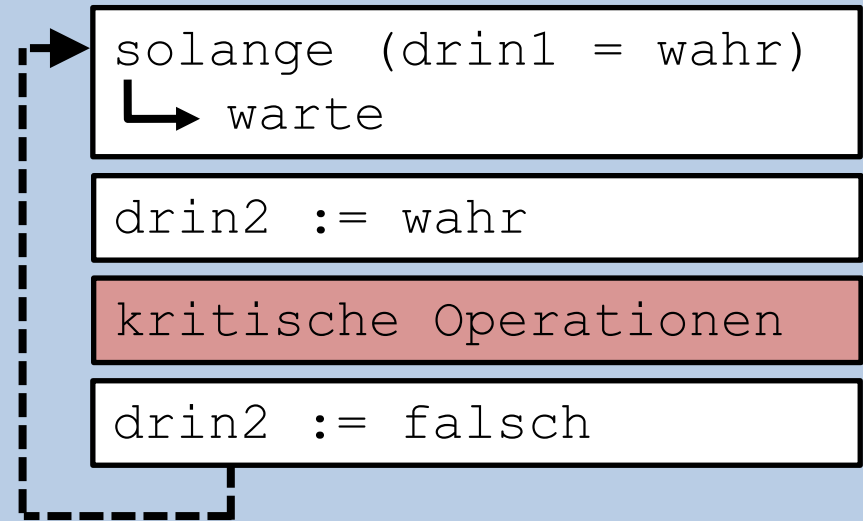
Softwarelösung – Verbesserungsversuch

Invertierung des vorherigen Ansatzes: Statt Zugangsvariable eine **Sperrvariable je Prozess**, die angibt, ob sich der jeweilige Prozess im kritischen Abschnitt befindet:

Prozess 1



Prozess 2



Softwarelösung – Verbesserungsversuch

Hinweis: `drin1` und `drin2` werden mit `falsch` initialisiert.

Prozess 1

solange (`drin2 = wahr`)
└─ warte

`drin1 := wahr`

kritische Operationen

`drin1 := falsch`

Prozess 2

solange (`drin1 = wahr`)
└─ warte

`drin2 := wahr`

kritische Operationen

`drin2 := falsch`

Softwarelösung – Verbesserungsversuch

Probleme: 1. immer noch beschäftigtes Warten (*busy wait*)

Prozess 1

solange (drin2 = wahr)
└→ warte

drin1 := wahr

kritische Operationen

drin1 := falsch

Prozess 2

solange (drin1 = wahr)
└→ warte

drin2 := wahr

kritische Operationen

drin2 := falsch

Softwarelösung – Verbesserungsversuch

- Probleme:**
1. immer noch beschäftigtes Warten (*busy wait*)
 2. Unterbrechung kann zu Fehlverhalten führen

Um dies zu verdeutlichen, wird im Folgenden ein mögliches Szenario schrittweise analysiert.

Verbesserungsversuch – Analyse

Ausgangssituation: Beide Prozesse gerade gestartet
Prozess 1 hat die CPU

Prozess 1

Prozess 2

Verbesserungsversuch – Analyse

- **Prozess 1** prüft, ob `drin2` den Wert `wahr` hat.
- Da `drin1` und `drin2` mit `falsch` initialisiert wurden, muss Prozess 1 nicht warten und verlässt die Schleife.
- Zufällig ist jetzt die **Zeitscheibe** des Prozesses abgelaufen.

Prozess 1

```
solange (drin2 = wahr)  
└─ warte
```

Prozess 2

Verbesserungsversuch – Analyse

- **Folge:** Es findet ein **Prozesswechsel** auf **Prozess 2** statt.
- Dieser prüft, ob `drin1` den Wert `wahr` hat.
- Da Prozess1 diesen Wert noch **nicht anpassen** konnte, muss auch Prozess 2 nicht warten und verlässt die Schleife.

Prozess 1

```
solange (drin2 = wahr)  
└─ warte
```

Prozess 2

```
solange (drin1 = wahr)  
└─ warte
```

Verbesserungsversuch – Analyse

- Prozess 2 setzt den Wert von `drin2` auf `wahr` und betritt den **kritischen Abschnitt**.
- Zufällig läuft die **Zeitscheibe** von Prozess 2 ab, bevor er den kritischen Abschnitt verlassen hat.

Prozess 1

```
solange (drin2 = wahr)  
└─ warte
```

Prozess 2

```
solange (drin1 = wahr)  
└─ warte
```

```
drin2 := wahr
```

```
kritische Operationen
```

Verbesserungsversuch – Analyse

- **Prozess 1** wurde **hinter** der Warteschleife unterbrochen und hat von der veränderten Situation nichts bemerkt.
- Er setzt `drin1` auf `wahr` und betritt ebenfalls den **kritischen Abschnitt**.

Prozess 1

```
solange (drin2 = wahr)  
└─ warte
```

```
drin1 := wahr
```

```
kritische Operationen
```

Prozess 2

```
solange (drin1 = wahr)  
└─ warte
```

```
drin2 := wahr
```

```
kritische Operationen
```

Verbesserungsversuch – Analyse

Resultat: Beide Prozesse sind im kritischen Abschnitt!

Prozess 1

```
solange (drin2 = wahr)  
└─ warte
```

```
drin1 := wahr
```

kritische Operationen

Prozess 2

```
solange (drin1 = wahr)  
└─ warte
```

```
drin2 := wahr
```

kritische Operationen



Verbesserungsversuch – Analyse

Resultat: Beide Prozesse sind im kritischen Abschnitt!

Ursache: Prüfen und Setzen von `drin1` und `drin2` sind zwei Einzeloperationen, zwischen denen eine Unterbrechung stattfinden kann!

Prozess 1

```
solange (drin2 = wahr)  
└─ warte
```

```
drin1 := wahr
```

```
kritische Operationen
```

```
drin1 := falsch
```

Prozess 2

```
solange (drin1 = wahr)  
└─ warte
```

```
drin2 := wahr
```

```
kritische Operationen
```

```
drin2 := falsch
```



Zweiter Verbesserungsversuch

Lösungsansatz: Vertauschen des Setzens und Prüfens.

Ist das die Lösung? Überlegen Sie selbst für einen Moment!

Prozess 1

drin1 := wahr

**solange (drin2 = wahr)
↳ warte**

kritische Operationen

drin1 := falsch

Prozess 2

drin2 := wahr

**solange (drin1 = wahr)
↳ warte**

kritische Operationen

drin2 := falsch

Zweiter Verbesserungsversuch – Analyse

Annahme: Nach dem Start hat Prozess 1 gerade Gelegenheit bekommen, seine Eintrittsvariable zu setzen. Danach wurde er unterbrochen und Prozess 2 ist dran.

Prozess 2 setzt ebenfalls seine Eintrittsvariable.

Prozess 1

```
drin1 := wahr
```

Prozess 2


```
drin2 := wahr
```


Zweiter Verbesserungsversuch – Analyse

Da die Eintrittsvariable von Prozess 1 jedoch gesetzt ist, kann Prozess 2 die Warteschleife nicht überwinden.

Prozess 1

```
drin1 := wahr
```



Prozess 2

```
drin2 := wahr
```

```
solange (drin1 = wahr)  
  ↳ warte
```

Zweiter Verbesserungsversuch – Analyse

Da die Eintrittsvariable von Prozess 1 jedoch gesetzt ist, kann Prozess 2 die Warteschleife nicht überwinden.

Dasselbe gilt nach Prozesswechsel auch für Prozess 1!

Prozess 1

```
drin1 := wahr
```

```
solange (drin2 = wahr)  
└─ warte
```

Prozess 2

```
drin2 := wahr
```

```
solange (drin1 = wahr)  
└─ warte
```



Zweiter Verbesserungsversuch – Analyse

Resultat: Beide Prozesse warten darauf, dass der jeweils andere seine Eintrittsvariable wieder auf `false` setzt. Die Prozesse **blockieren** sich gegenseitig!

Prozess 1

```
drin1 := wahr
```

```
solange (drin2 = wahr)  
└─ warte
```

Prozess 2

```
drin2 := wahr
```

```
solange (drin1 = wahr)  
└─ warte
```

Gibt es überhaupt eine rein softwarebasierte Lösung für das Problem des gegenseitigen Ausschlusses?



Algorithmus von Peterson

Grundidee: Zunächst bekundet ein Prozess **Interesse**, den kritischen Abschnitt zu betreten. Danach gibt er dem **anderen** Prozess die Möglichkeit, auf dieses Interesse zu reagieren.

Anders als bei den vorangegangenen Ansätzen versucht ein Prozess also nicht, „gierig“ die Zulassung zum kritischen Abschnitt zu erhalten, sondern gibt vorher die **Kontrolle** noch einmal ab.

Algorithmus von Peterson

Mit `int1/int2` bekunden die Prozesse ihr Interesse, den kritischen Abschnitt zu betreten.

Prozess 1

int1 := wahr

dran := 2

solange (`dran = 2`) UND
 (`int2 = wahr`)
└─ warte

kritische Operationen

int1 := falsch

Prozess 2

int2 := wahr

dran := 1

solange (`dran = 1`) UND
 (`int1 = wahr`)
└─ warte

kritische Operationen

int2 := falsch

Algorithmus von Peterson

dran gibt an, wer die **Warteschleife überwinden** kann, sofern **beide** Prozesse Interesse haben. Es „gewinnt“ hier stets der Prozess, der **zuerst** den Wert für dran gesetzt hat.

Prozess 1

int1 := wahr

dran := 2

solange (dran = 2) UND
 (int2 = wahr)
└─ warte

kritische Operationen

int1 := falsch

Prozess 2

int2 := wahr

dran := 1

solange (dran = 1) UND
 (int1 = wahr)
└─ warte

kritische Operationen

int2 := falsch

Algorithmus von Peterson

Interessiert sich der jeweils andere Prozess **nicht** für den kritischen Abschnitt, wird die Warteschleife sofort überwunden.

Prozess 1

int1 := wahr

dran := 2

solange (dran = 2) **UND**
└ (int2 = wahr)
└ warte

kritische Operationen

int1 := falsch

Prozess 2

int2 := wahr

dran := 1

solange (dran = 1) **UND**
└ (int1 = wahr)
└ warte

kritische Operationen

int2 := falsch

Algorithmus von Peterson

- **Verallgemeinerung** auf N Prozesse ist bei Inkaufnahme höheren Aufwandes möglich,
- **beschäftigtes Warten** bleibt bestehen

Atomare Operationen

- Softwarelösungen können wesentlich einfacher implementiert werden, wenn die **Unterbrechung** zwischen Lesen und Setzen der Zutrittsvariablen sicher verhindert werden kann.
- Zwei **Varianten**:
 1. Interrupts **blockieren** → Nur im Kernel-Modus möglich, daher nicht für Nutzerprogramme geeignet,
 2. Unterstützung durch Hardware mittels geeigneter **atomarer** (ununterbrechbarer) **Operationen**
- Bei Intel-CPU's gibt es dafür u.a. die atomare Operation **CMPXCHG** (vergleichen und Wert austauschen). Damit sind Softwareimplementierungen wesentlich leichter.
- **Busy Wait** bleibt!

Zusammenfassung Softwarelösungen

- Gegenseitiger Ausschluss ist notwendig bei Vorhandensein kritischer Abschnitte und konkurrender Prozesse,
- **Softwarelösung** für gegenseitigen Ausschluss ist möglich, hat jedoch zwei Probleme:
 1. **Umständlich**, weil nicht unterbrechbares Prüfen und Setzen von Variablen softwarebasiert nicht möglich,
 2. Ressourcenintensiv durch **beschäftigtes Warten**
- Hardware kann mit **atomaren Operationen** unterstützen, die jedoch nicht das beschäftigte Warten vermeiden.