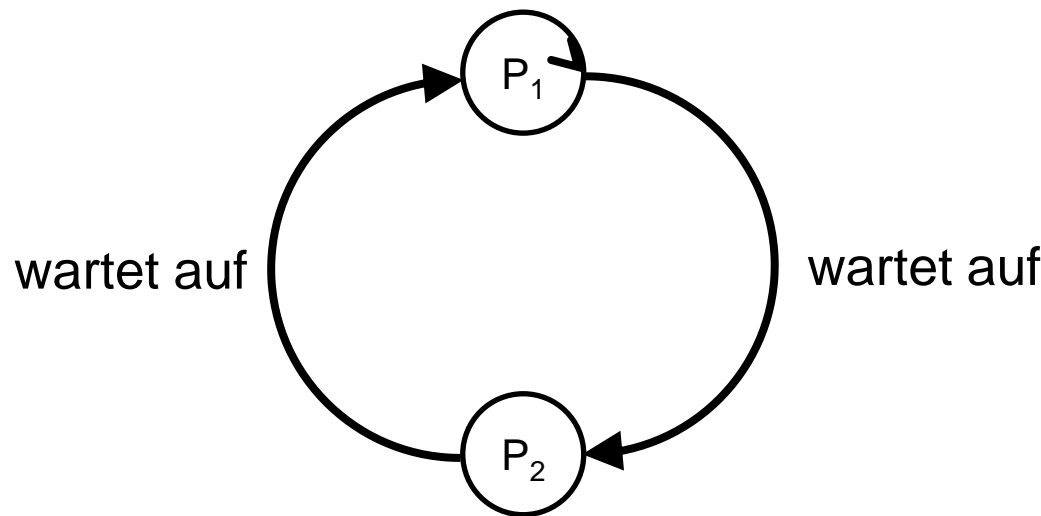


Prozesssynchronisation III

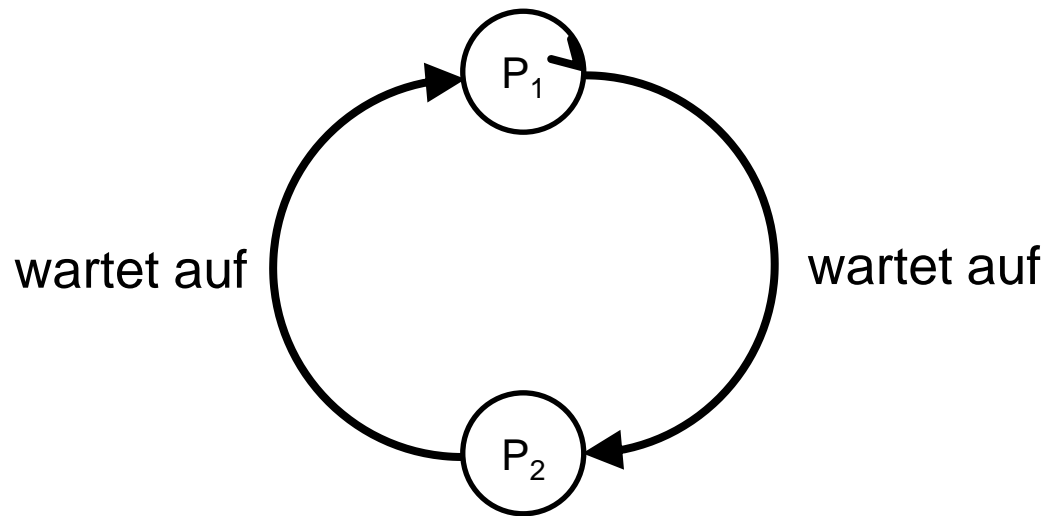
Verklemmungen (Deadlocks)

Als **Verklemmung** oder **Deadlock** wird eine Situation bezeichnet, in der **mindestens zwei** Prozesse nicht weiterarbeiten können, weil sie gegenseitig auf eine jeweils vom anderen zu erteilende **Freigabe** warten.



Verklemmungen (Deadlocks)

Verklemmungen können bei der Verwendung von Semaphoren auftreten, sind aber nicht darauf beschränkt. Grundsätzlich ist **jeder Mechanismus** zur Synchronisation nebenläufiger Prozesse, der **beliebig langes Warten** auf andere Prozesse oder Ressourcen zulässt, verklemmungsgefährdet.



Verklemmungen – Beispiel

Beispiel: Verklemmungsgefährdete Implementierung des Erzeuger-Verbraucher-Problems durch **einseitiges** Vertauschen der Semaphorenpassagen.

Erzeuger

P(frei)

P(mutex)

Element einfügen

V(mutex)

V(belegt)

Verbraucher

P(belegt)

P(mutex)

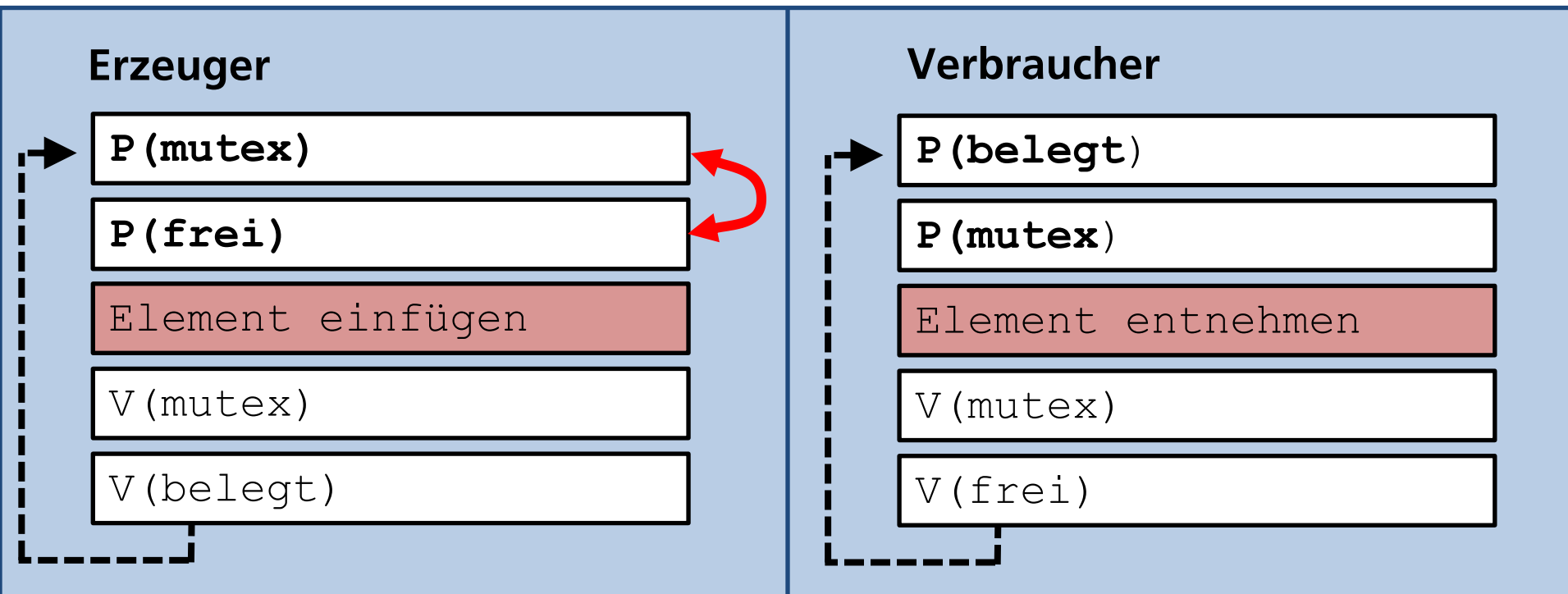
Element entnehmen

V(mutex)

V(frei)

Verklemmungen – Beispiel

Beispiel: Verklemmungsgefährdete Implementierung des Erzeuger-Verbraucher-Problems durch **einseitiges** Vertauschen der Semaphorenpassagen.



Verklemmungen – Beispiel

Bei vollem Puffer muss der Erzeuger an der Passage von `frei` darauf warten, dass der Verbraucher `V(frei)` aufruft.

Erzeuger

`P(mutex)`

`P(frei)`

Verbraucher

Verklemmungen – Beispiel

Bei vollem Puffer muss der Erzeuger an der Passage von `frei` darauf warten, dass der Verbraucher `V(frei)` aufruft. An dieser Stelle blockiert der Erzeuger.

Erzeuger

`P(mutex)`

`P(frei) → blockiert`

Verbraucher

Verklemmungen – Beispiel

Der Verbraucher kann zwar den Semaphor belegt passieren, allerdings hält der Erzeuger immer noch den Semaphor `mutex`.

Erzeuger

P (`mutex`)

P (`frei`) → blockiert

Verbraucher

P (**belegt**)

P (`mutex`)

Verklemmungen – Beispiel

Der Verbraucher kann zwar den Semaphor belegt passieren, allerdings hält der Erzeuger immer noch den Semaphor `mutex`. Der Verbraucher blockiert ebenfalls!

Erzeuger

`P(mutex)`

`P(frei) → blockiert`

Verbraucher

`P(belegt)`

`P(mutex) → blockiert`

Verklemmungen – Beispiel

Folge: Keiner der beiden Prozesse kann aus diesem Zustand herauskommen, sie sind verklemmt.

Erzeuger

P(mutex)

P(frei) → blockiert

Verbraucher

P(belegt)

P(mutex) → blockiert

Dinierende Philosophen



the dining philosophers problem

Dinierende Philosophen

Die Problematik der Verklemmungen wird gerne an dem von Dijkstra eingeführten Gedankenspiel der **dinierenden Philosophen** erläutert:



the d i n i n g
philosophers problem

Dinierende Philosophen

Die Problematik der Verklemmungen wird gerne an dem von Dijkstra eingeführten Gedankenspiel der **dinierenden Philosophen** erläutert:

5 Philosophen sitzen zum Dinner gemeinsam an einem runden Tisch. Jeder Philosoph hat einen Teller mit seinem Essen vor sich stehen. Es wird mit Stäbchen gegessen, aber leider hat der Hausherr nur jeweils ein Stäbchen pro Person.



Dinierende Philosophen

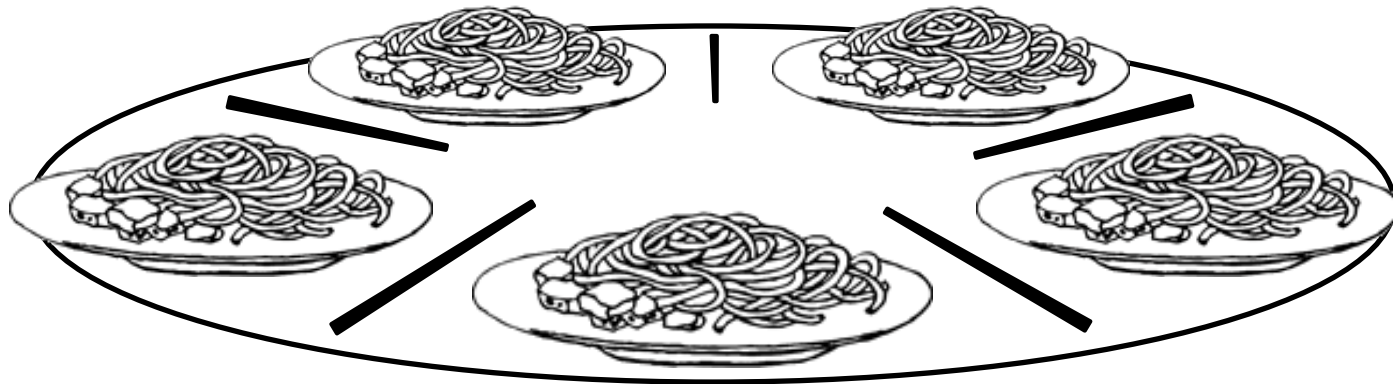
Die Problematik der Verklemmungen wird gerne an dem von Dijkstra eingeführten Gedankenspiel der **dinierenden Philosophen** erläutert:

5 Philosophen sitzen zum Dinner gemeinsam an einem runden Tisch. Jeder Philosoph hat einen Teller mit seinem Essen vor sich stehen. Es wird mit Stäbchen gegessen, aber leider hat der Hausherr nur jeweils ein Stäbchen pro Person.

Ein Philosoph, der essen möchte, greift zuerst nach dem Stäbchen zu seiner Linken und dann nach dem Stäbchen zu seiner Rechten. Steht eines dieser Stäbchen nicht zur Verfügung, weil sein Nachbar es gerade in der Hand hält, wartet der Philosoph, bis es wieder zur Verfügung steht.

Dinierende Philosophen

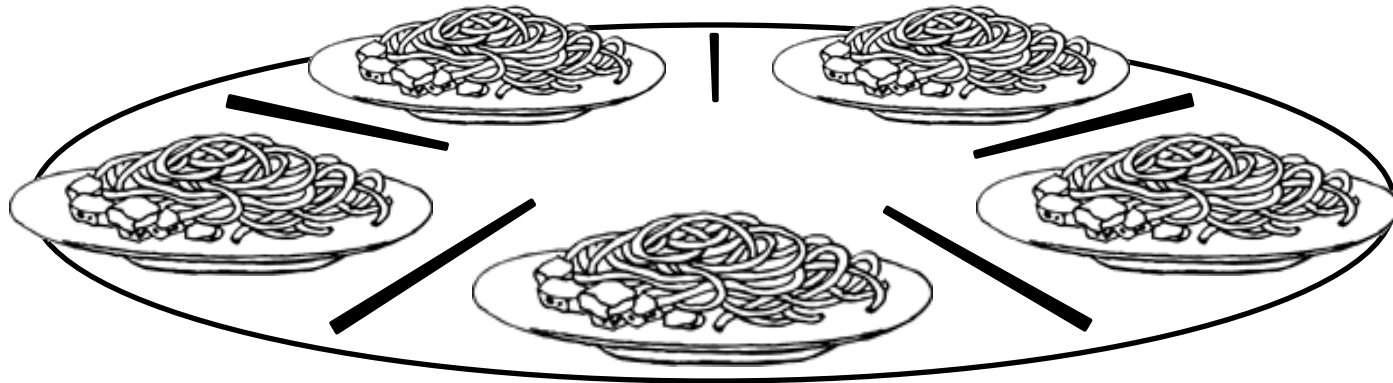
Solange nur einzelne Philosophen essen, ist dieses Vorgehen unproblematisch. Was aber geschieht, wenn alle Philosophen **exakt gleichzeitig** anfangen wollen, zu essen?



Dinierende Philosophen

In diesem Fall greift jeder Philosoph das Stäbchen zu seiner Linken und wartet dann unendlich lange darauf, dass das Stäbchen zu seiner Rechten verfügbar wird.

Folge: Die Philosophen verhungern!



Mögliche Lösung: Hierarchie einführen

- Die Stäbchen werden aufsteigend durchnummeriert und **sortiert** auf dem Tisch verteilt (bspw. im Uhrzeigersinn).
- Will ein Philosoph essen, muss er stets das Stäbchen mit der **niedrigeren** Nummer zuerst aufnehmen.
- Das Stäbchen mit der **höchsten** Nummer kann nur von dem einen Philosophen aufgenommen werden, zu dessen Linken es liegt (Sortierung im Uhrzeigersinn vorausgesetzt), er kann also auf jeden Fall essen.

Folge: Die Verklemmung ist vermieden.

Voraussetzungen für Verklemmungen

Notwendige und hinreichende Bedingungen

1. **Beschränkte** Belegung (*mutual exclusion*)
Zugriff auf Ressourcen wird mittels Reservierung synchronisiert.
2. **Zusätzliche** Belegung (*hold-and-wait*)
Ressourcen werden gehalten, bis alle Ressourcen zur Verfügung stehen.
3. **Keine** vorzeitige Rückgabe (*no preemption*)
Bereits reservierte Ressourcen werden gehalten, auch wenn auf weitere Ressourcen gewartet wird.

Dies ermöglicht: Gegenseitiges Warten (*circular wait*)
und damit eine Verklemmung.

Umgang mit Verklemmungen

Möglicher Umgang mit Verklemmungen:

1. das Problem **ignorieren**,

oder

2. Verklemmungen **erkennen** und beseitigen,

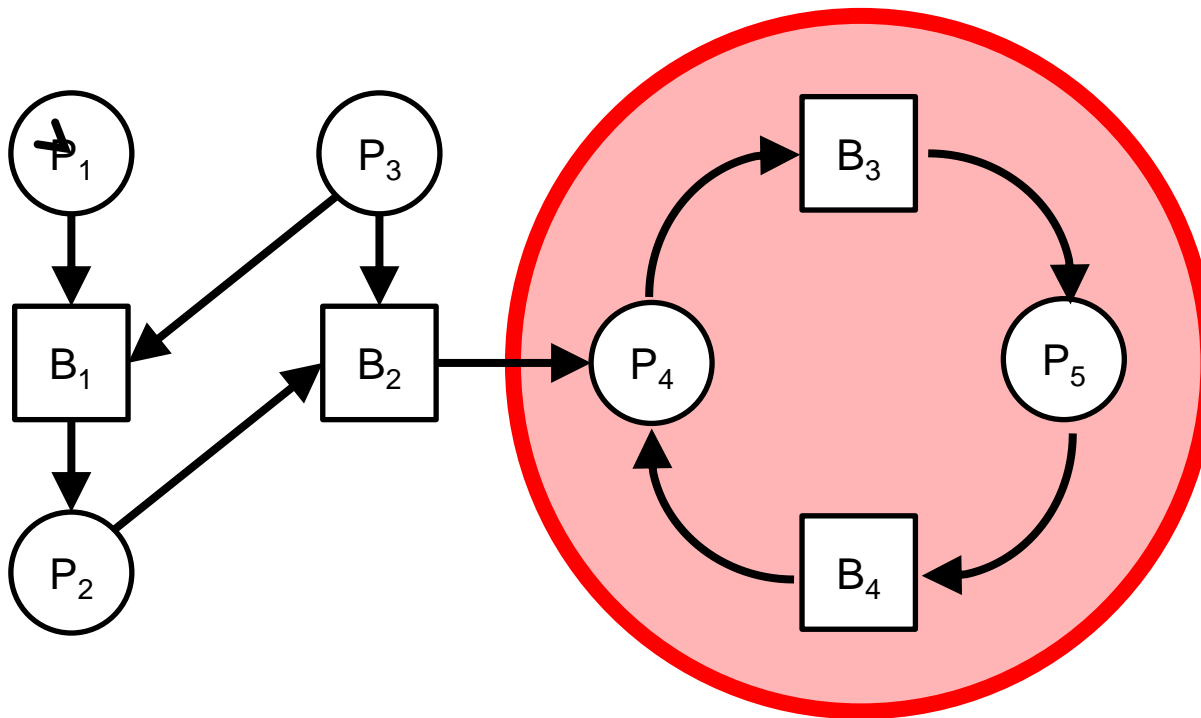
3. Verklemmungen vor der Entstehung **vermeiden**,

4. Verklemmungen **ausschließen (unmöglich machen)**, indem eine der vorgenannten Bedingungen (1)–(3) verhindert wird

Verklemmungen erkennen

- Anzeichen:**
1. viele Prozesse warten, obwohl die CPU nicht ausgelastet ist
 2. Prozesse warten (ungewöhnlich) lange

Analyse des Betriebsmittelgraphen:



Zyklus bedeutet Verklemmung

Verklemmungen beseitigen

Um eine erkannte Verklemmung zu beseitigen, stehen im Wesentlichen drei Möglichkeiten zur Verfügung:

1. Prozesse **abbrechen**: Dies ist das sicherste Mittel, eine Verklemmung aufzulösen. Dieser Ansatz kann allerdings zu undefinierten Zuständen in der Anwendung führen, zu der der betreffende Prozess gehört.
2. Zu einem **früheren Prozesszustand** (Checkpoint) zurückkehren: Dieser Ansatz ist aus Datenbankanwendungen bekannt und dort sehr erfolgreich. Nachteil: Checkpoints müssen verwaltet werden und sind nicht immer möglich.
3. Reservierte **Betriebsmittel** wieder **entziehen**: Dies kann die zyklische Abhängigkeit der Verklemmung auflösen. Problem: Prozesse können verhungern, weil ihnen immer wieder die Betriebsmittel entzogen werden.

Verklemmungen vermeiden

Idee: Vor der Betriebsmittelreservierung wird geprüft, ob diese zu einer Verklemmung führen kann. In diesem Fall wird die Reservierungsanfrage **abgelehnt**.

- **Bankieralgorithmus** nach Dijkstra:
 - Prüfe, ob eine neue Reservierungsanforderung so erfüllt werden kann, dass trotzdem die bereits laufenden Prozesse ihre Arbeit erledigen können.
 - Analogie zu (konservativem) Bankier: Gib nur Kreditzusagen, wenn Du sie auch erfüllen kannst, ohne die Zusagen an andere Kunde zurücknehmen zu müssen.

Verklemmungen vermeiden

- Der Bankieralgorithmus erwartet allerdings, dass **alle** Prozesse **vor** dem Start angeben können, welche Betriebsmittel sie benötigen und dass auch die Menge der Betriebsmittel **vorab bekannt** ist.
- Weiterhin ist der Algorithmus sehr aufwändig (er kostet Zeit). In der **Praxis** wird dieses Verfahren daher nur **selten** eingesetzt.

Verklemmungen ausschließen

Ursachen für Verklemmungen direkt adressieren:

1. Beschränkte Belegung (*mutual exclusion*)

Mögliche Lösung: Verwaltung aller Betriebsmittel durch einen einzelnen Serviceprozess.

2. Zusätzliche Belegung (*hold-and-wait*)

Mögliche Lösung: Nur ein Betriebsmittel je Prozess.

3. Keine vorzeitige Rückgabe (*no preemption*)

Mögliche Lösung: Ähnlich wie bei der Verklemmungsvermeidung (Prozess abbrechen, Betriebsmittel entziehen, etc.)

Zwischenfazit Verklemmungen

- Verklemmungen sind eine **Begleiterscheinung** der **Ressourcenallokation** durch Reservieren und Warten.
- Sie können auftreten, wenn mehrere Prozesse für die Bearbeitung ihrer Aufgaben mehrere **gegenseitig benötigte** Ressourcen reservieren wollen.
- Es gibt Mechanismen zur **Erkennung, Auflösung, Vermeidung** und zum **Ausschluss** von Verklemmungen. Diese führen z.T. in der Praxis jedoch zu neuen Problemen.

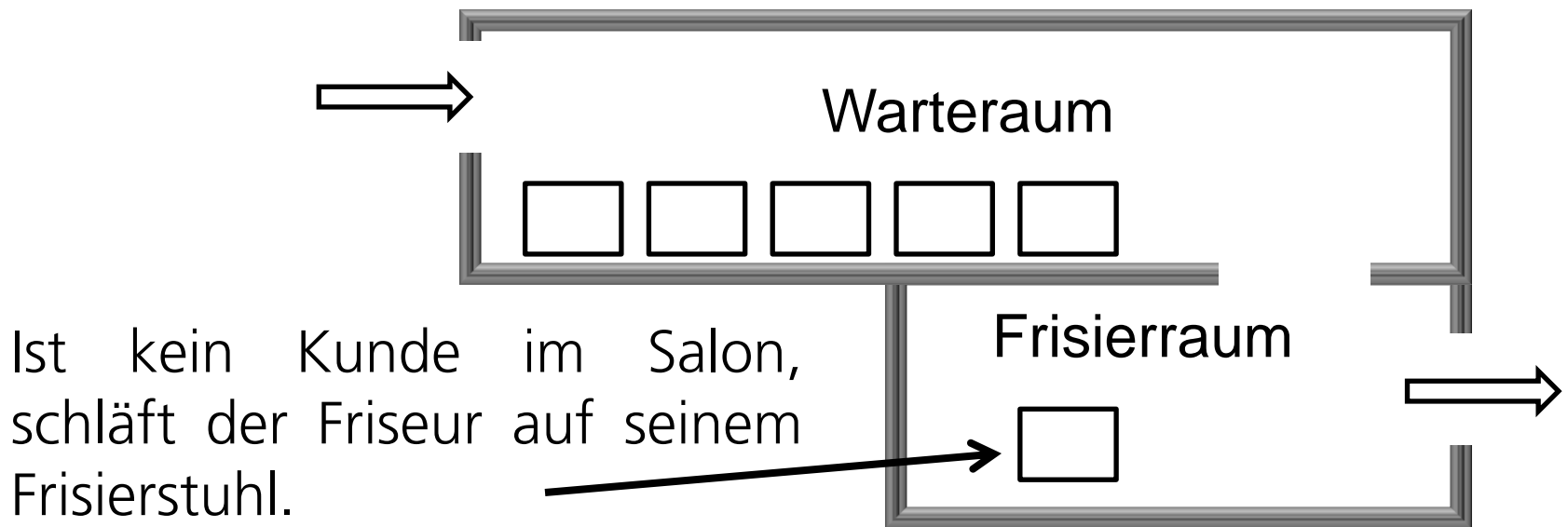
Der schlafende Friseur

Der **schlafende Friseur** beschreibt eine Synchronisationsaufgabe für Warteschlangen mit **begrenzter Kapazität**.

Die Fragestellung ist der des Erzeuger-Verbraucher-Problems ähnlich, allerdings soll nur der **Bearbeiter** (Verbraucher) dauerhaft blockiert werden können. Eingehende Prozesse, die nicht bearbeitet und auch nicht in die Warteschlange eingereiht werden können, blockieren nicht, sondern werden **abgewiesen**.

Der schlafende Friseur

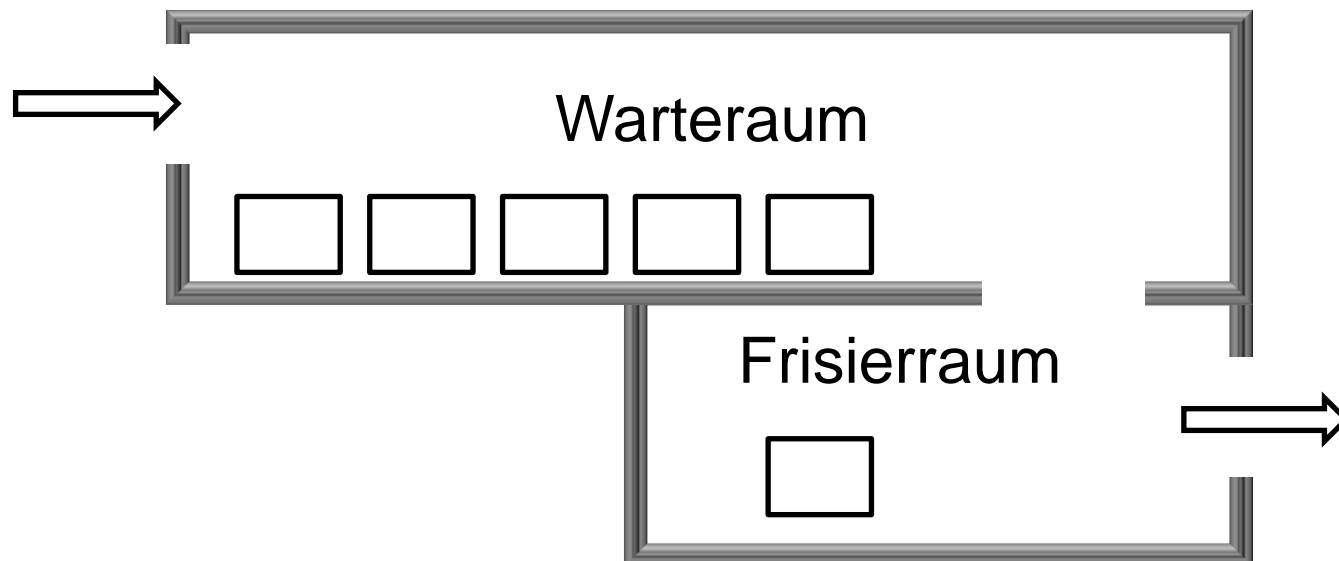
Ein Friseursalon besteht aus **zwei Zimmern**, die nicht direkt voneinander einsehbar sind. Im Hinterzimmer bedient der Friseur seine Kunden, während im Vorderzimmer eine begrenzte Anzahl von Stühlen für wartende Kunden zur Verfügung steht.



Der schlafende Friseur

Betritt ein Kunde den Salon und

1. der Friseur **schläft**, so **weckt** der Kunde ihn,
2. der Friseur **arbeitet**, so versucht der Kunde, im **Warteraum** Platz zu nehmen,
3. der Warteraum ist **voll**, so **verlässt** der Kunde den Salon.



Ansatz mit Semaphoren

Erster Ansatz: Ähnlich wie beim Erzeuger-Verbraucher-Problem wird ein Semaphor für die belegten Plätze im Warteraum verwendet. Da die Warteschlange verändert werden kann, stellt der Zugriff darauf einen kritischen Abschnitt dar, für den **gegenseitiger Ausschluss** gewährleistet sein muss. Weiterhin muss eine Möglichkeit geschaffen werden, die **freien Plätze** ohne Blockade zu modellieren.

Problem: Der Wert eines Semaphors kann nicht **direkt** abgefragt werden, schließlich stehen nur die Funktion $P()$ und $V()$ zur Verfügung.

Lösung: Die Verwaltung der freien Plätze erfolgt mit einer „gewöhnlichen“ Variable. Der Zugriff auf diese Variable wird mit gegenseitigem Ausschluss geschützt.

Lösung als Übungsaufgabe!

Die Readers-Writers-Probleme

Bisher haben wir angenommen, dass der Zugriff auf Ressourcen, die von mindestens einem Prozess verändert werden können, stets durch gegenseitigen Ausschluss gesichert wird.

Tatsächlich aber muss dies nur im Falle eines **verändernden** Zugriffs geschehen. Greifen **alle** Prozesse zu einer bestimmten Zeit nur **lesend** auf die Ressource zu, wird kein gegenseitiger Ausschluss benötigt.

Die folgenden Fragestellungen und deren Lösungen adressieren also die **Performanz** des Zugriffs auf gemeinsam genutzte Ressourcen.

Das erste Readers-Writers-Problem

Voraussetzung: Es soll sowohl lesender als auch schreibender Zugriff auf eine gemeinsam genutzte Ressource synchronisiert werden. Die Anzahl der Prozesse, die die Ressource verwenden, ist nicht bekannt.

Aufgabe: Wird die Ressource aktuell nur lesend verwendet und ein weiterer Prozess will ebenfalls nur lesend auf diese Ressource zugreifen, darf dieser **nicht blockiert** werden (*readers preference*). Im Falle **schreibenden** Zugriffs muss jedoch weiterhin gegenseitiger **Ausschluss** auch für lesende Prozesse gewährleistet sein.

Das erste Readers-Writers-Problem

Lösungsansatz:

- Die **Anzahl** der z.Zt. **lesenden** Prozesse wird in einer durch gegenseitigen Ausschluss gesicherten Variablen `z` hinterlegt.
- Schreibende Prozesse müssen **stets** einen Semaphor `schreiben` passieren.
- Versucht ein Prozess als bisher **einziger, lesend** auf die Ressource zuzugreifen, muss er ebenfalls den Semaphor `schreiben` passieren. Alle nachfolgenden Leseprozesse müssen diesen Semaphor nicht mehr passieren, solange noch mindestens ein anderer Prozess liest.
- Sind **alle Leseprozesse beendet**, wird der Semaphor `schreiben` wieder verlassen.

Initialisierung:

```
mutex, schreiben    : Semaphor;  
z                   : Ganzzahl;
```

```
mutex.wert := schreiben.wert := 1;  
z := 0;
```

Leseprozess

```
P(mutex);  
z := z + 1;  
Ist z = 1  
    Ja: P(schreiben);  
V(mutex);  
  
// Daten lesen  
  
P(mutex);  
z := z - 1;  
Ist z = 0  
    Ja: V(schreiben);  
V(mutex);
```

Schreibprozess

```
P(schreiben);  
  
// Daten schreiben  
  
V(schreiben)
```

Das erste Readers-Writers-Problem

Problem: Lesende Prozesse werden so stark bevorzugt, dass wartende schreibende Prozesse stets verdrängt werden. Kommen immer wieder neue Leseprozesse hinzu, wartet ein schreibender Prozess im schlechtesten Fall für immer.

Das zweite Readers-Writers-Problem

Erweiterung der Aufgabenstellung (das zweite Readers-Writers-Problem)

Schreibende Prozesse dürfen nicht länger warten als unbedingt notwendig (*writers preference*). Lesende Prozesse sollen weiterhin parallel zugreifen können, solange kein Prozess schreibt.

Das zweite Readers-Writers-Problem

Ansatz:

1. Alle Leseprozesse und der erste neu eintretende Schreibprozess müssen einen gemeinsamen Semaphor `warteschlange` passieren. Jeder Leseprozess und der letzte wartende Schreibprozess geben diesen Semaphor am Ende ihrer Arbeit wieder frei. So wird sichergestellt, dass neu eintreffende Leseprozesse einen wartenden Schreibprozess nicht überholen können.
2. Durch einen **Engpassemaphor** bei den Leseprozessen wird sichergestellt, dass nicht mehrere Leser hintereinander auf den Warteschlangensemaphor warten. Dadurch kommt der Schreiber so schnell wie möglich an die Reihe.

Das zweite Readers-Writers-Problem

Der Warteschlangensemaphor sorgt für eine Einreihung der Lese- und Schreibprozesse.

Initialisierung:

```
warteschlange, schreiben, engpass,  
lmutex, smutex  : Semaphor;  
lz, sz          : Ganzzahl;
```

```
lesen.wert      := 1;  
schreiben.wert  := 1;  
engpass.wert    := 1;  
lmutex.wert     := 1;  
smutex.wert     := 1;  
lz := sz := 0;
```

Das zweite Readers-Writers-Problem

Es darf nur ein Prozess zur Zeit schreibend zugreifen, daher wird ein Semaphor schreiben für den gegenseitigen Ausschluss benötigt.

Initialisierung:

```
warteschlange, schreiben, engpass,  
lmutex, smutex : Semaphor;  
lz, sz          : Ganzzahl;
```

```
lesen.wert      := 1;  
schreiben.wert  := 1;  
engpass.wert    := 1;  
lmutex.wert     := 1;  
smutex.wert     := 1;  
lz := sz := 0;
```

Das zweite Readers-Writers-Problem

Der Semaphor `engpass` sichert, dass nur **ein Leseprozess** zur Zeit auf den Lesesemaphor warten kann. Dadurch können Schreibprozesse, die in diesem Ansatz ebenfalls wie Leser behandelt werden, schnellstmöglich an die Reihe kommen.

Initialisierung:

```
warteschlange, schreiben, engpass,  
lmutex, smutex    : Semaphor;  
lz, sz            : Ganzzahl;
```

```
lesen.wert        := 1;  
schreiben.wert    := 1;  
engpass.wert      := 1;  
lmutex.wert       := 1;  
smutex.wert       := 1;  
lz := sz := 0;
```


Das zweite Readers-Writers-Problem

Die Semaphoren `lmutex` und `rmutex` dienen dem gegenseitigen Ausschluss bei der Bearbeitung der Zähler.

Initialisierung:

```
warteschlange, schreiben, engpass,  
lmutex, smutex : Semaphor;  
lz, sz          : Ganzzahl;
```

```
lesen.wert      := 1;  
schreiben.wert  := 1;  
engpass.wert    := 1;  
lmutex.wert     := 1;  
smutex.wert     := 1;  
lz := sz := 0;
```

Das zweite Readers-Writers-Problem

Die Zähler `lz` und `sz` speichern die aktuelle Anzahl parallel lesender bzw. wartender schreibender Prozesse.

Initialisierung:

```
warteschlange, schreiben, engpass,  
lmutex, smutex  : Semaphor;  
lz, sz          : Ganzzahl;
```

```
lesen.wert      := 1;  
schreiben.wert  := 1;  
engpass.wert    := 1;  
lmutex.wert     := 1;  
smutex.wert     := 1;  
lz := sz := 0;
```

Das zweite Readers-Writers-Problem

Alle Semaphoren werden mit dem Wert 1 initialisiert, die Zähler mit 0.

Initialisierung:

```
warteschlange, schreiben, engpass,  
lmutex, smutex   : Semaphor;  
lz, sz           : Ganzzahl;
```

```
warteschlange.wert := 1;  
schreiben.wert := 1;  
engpass.wert := 1;  
lmutex.wert := 1;  
smutex.wert := 1;  
lz := sz := 0;
```

Leseprozess

```
P(engpass);
P(warteschlange);
P(lmutex);
lz := lz + 1;
Ist lz = 1
    Ja: P(schreiben));
V(lmutex);
V(warteschlange);
V(engpass);

// Daten lesen

P(lmutex);
lz := lz - 1;
Ist lz = 0
    Ja: V(schreiben);
V(lmutex);
```

Schreibprozess

```
P(smutex);
sz := sz + 1;
Ist (sz = 1)
    Ja: P(warteschlange);
V(smutex);
P(schreiben);

// Daten schreiben

V(schreiben);
P(smutex);
sz := sz - 1;
Ist sz = 0
    Ja: V(warteschlange);
V(smutex);
```

Das zweite Readers-Writers-Problem – Eigenschaften

1. Der Ansatz setzt voraus, dass die Warteschlangen der Semaphoren **reihenfolgeerhaltend** sind, d.h. nach dem Prinzip *first-in-first-out* behandelt werden.
2. Treten ständig neue **schreibende** Prozesse auf, können diese die **lesenden** Prozesse **verdrängen**, weil die neu hinzugekommenen Schreibprozesse den Warteschlangen-semaphor **nicht** passieren müssen.
3. Der Ansatz beinhaltet viele Semaphoreaufrufe, es sind also viele **Kontextwechsel** (User-Mode → Kernel-Mode und zurück) notwendig. Gibt es geeignete **atomare Operationen** seitens der CPU (siehe *Prozess-synchronisation Teil I*), kann auf die Semaphoren `smutex` und `lmutex` verzichtet werden.

Das dritte Readers-Writers-Problem

Erweiterung der Aufgabenstellung (das dritte Readers-Writers-Problem)

Kein Prozess soll unbegrenzt lange auf seine Abarbeitung warten müssen.

Das dritte Readers-Writers-Problem

Ansatz: Die Warteschlange wird auf **alle Prozesse** ausgeweitet, also auch auf nachfolgende Schreibprozesse. Sie wird noch **vor Ausführung** der eigentlichen Lese- bzw. Schreiboperation wieder verlassen, allerdings mit einem kleinen Unterschied zwischen den Prozessen:

1. Lesende Prozesse verlassen diese Warteschlange direkt nach der Manipulation des Zählers wieder.
2. Schreibende Prozesse hingegen verlassen die Warteschlange erst, wenn sie die Schreiboperation ausführen dürfen. So verhindern sie, dass sie während des Wartens von Lesern überholt werden.

Ansatz mit Semaphoren

Implementierung als Übungsaufgabe!

Interprozesskommunikation

Gegenüber der Prozesssynchronisation ist die **Interprozesskommunikation** ein allgemeineres Problem, bei dem sowohl Aspekte der Synchronisation als der Übertragung von Nutzdaten relevant sind.

Wir behandeln an dieser Stelle überblicksartig:

- Signale
- Pipes
- Nachrichtensysteme

Signale sind von Nutzerprozessen erzeugbare **Nachrichten ohne Nutzlast**, die ein Ereignis oder eine Anforderung repräsentieren. Die übertragene Information besteht ausschließlich aus

- der Tatsache, dass ein Signal gesendet wurde (ein Ereignis eingetreten ist) und
- der Art des Signals.

Sie können als **Unterbrechungsanforderungen** zwischen **Nutzerprozessen** verstanden werden.

Mögliche Signale sind unter Anderem^{*)}:

- **SIGKILL:** Prozess umgehend beenden
(keine weiteren Aktionen durchführbar)
- **SIGTERM:** Beendigungsanfrage
(weitere Aktionen durchführbar)
- **SIGSTOP:** Prozess pausieren
- **SIGCONT:** Prozess fortführen
- **SIGUSR1:** Frei interpretierbar

^{*)} Dies ist nur eine kleine Auswahl am Beispiel von UNIX

Prozesse, die ein Signal empfangen, können:

1. eine **Standardreaktion** aufrufen (geschieht durch das Betriebssystem, muss nicht implementiert werden),
2. Eine **eigene Behandlungsroutine** aufrufen, um eine individuelle Reaktion zu ermöglichen.

Um diese individuelle Reaktion zu ermöglichen, wird die eigene Behandlungsroutine für das entsprechende Signal über einen Systemaufruf beim Betriebssystem registriert.

Dies ist nicht für alle Signale möglich (bspw. SIGKILL)!

Pipes

Jeder Prozess erhält bei seiner Erzeugung vom Betriebssystem einen Eingabe- und einen Ausgabekanal (Stream)

- Daten, die in den Ausgabekanal geschrieben werden (bspw. mittels `print()`, `write()`, etc.), werden vom Betriebssystem standardmäßig auf dem Bildschirm ausgegeben*),
- Eingabedaten von der Tastatur werden vom Betriebssystem an den Eingabekanal weitergeleitet *).

Die Steuerung der beiden Kanäle erfolgt über **Hardware-interrupts** (Eingabekanal) und **Systemrufe** (Ausgabekanal).

*) Die Wirklichkeit ist komplizierter, hier soll nur das Konzept erläutert werden

Umlenkung:

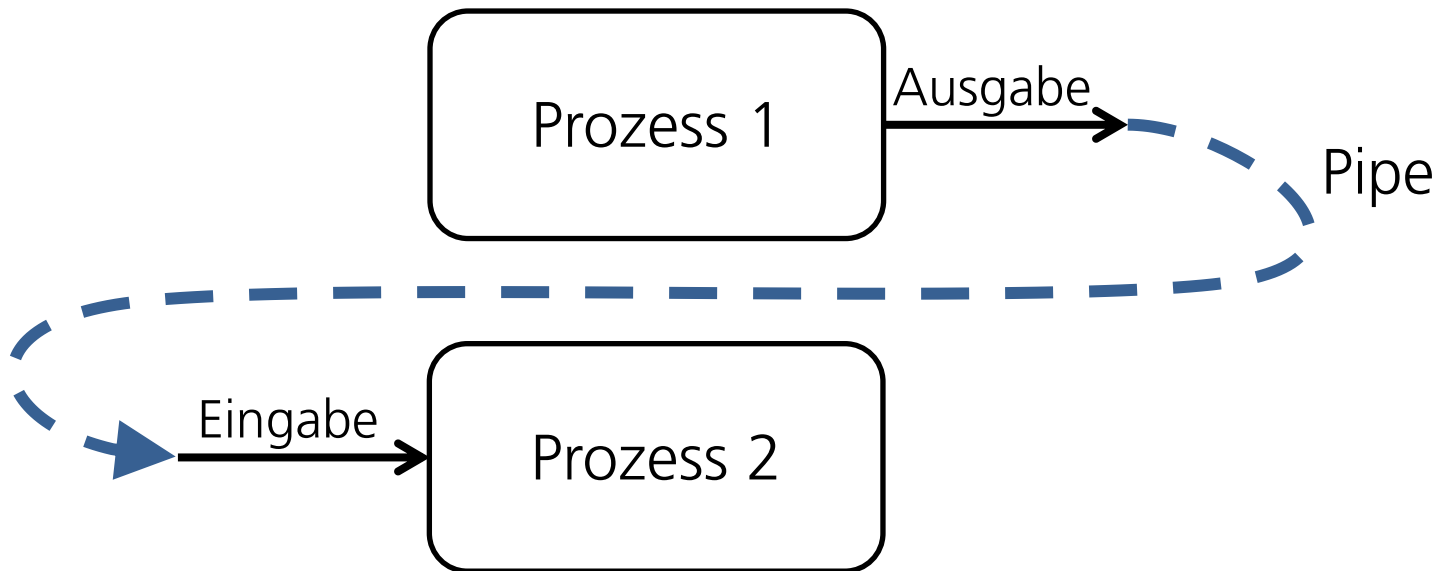
Das Betriebssystem kann Quelle bzw. Ziel der beiden Kanäle anderen Ressourcen zuordnen. Der Eingabekanal kann bspw. auch aus einer Datei heraus bedient werden und der Ausgabekanal kann in eine Datei schreiben.

Diese **Umlenkung** wird durch das Betriebssystem organisiert und ist für den Prozess vollständig transparent. Sie wird meistens vom Benutzer auf direkte oder indirekte Art per Systemaufruf angefordert.

Beispiel Linux: `ls -l > verzeichnis.txt`

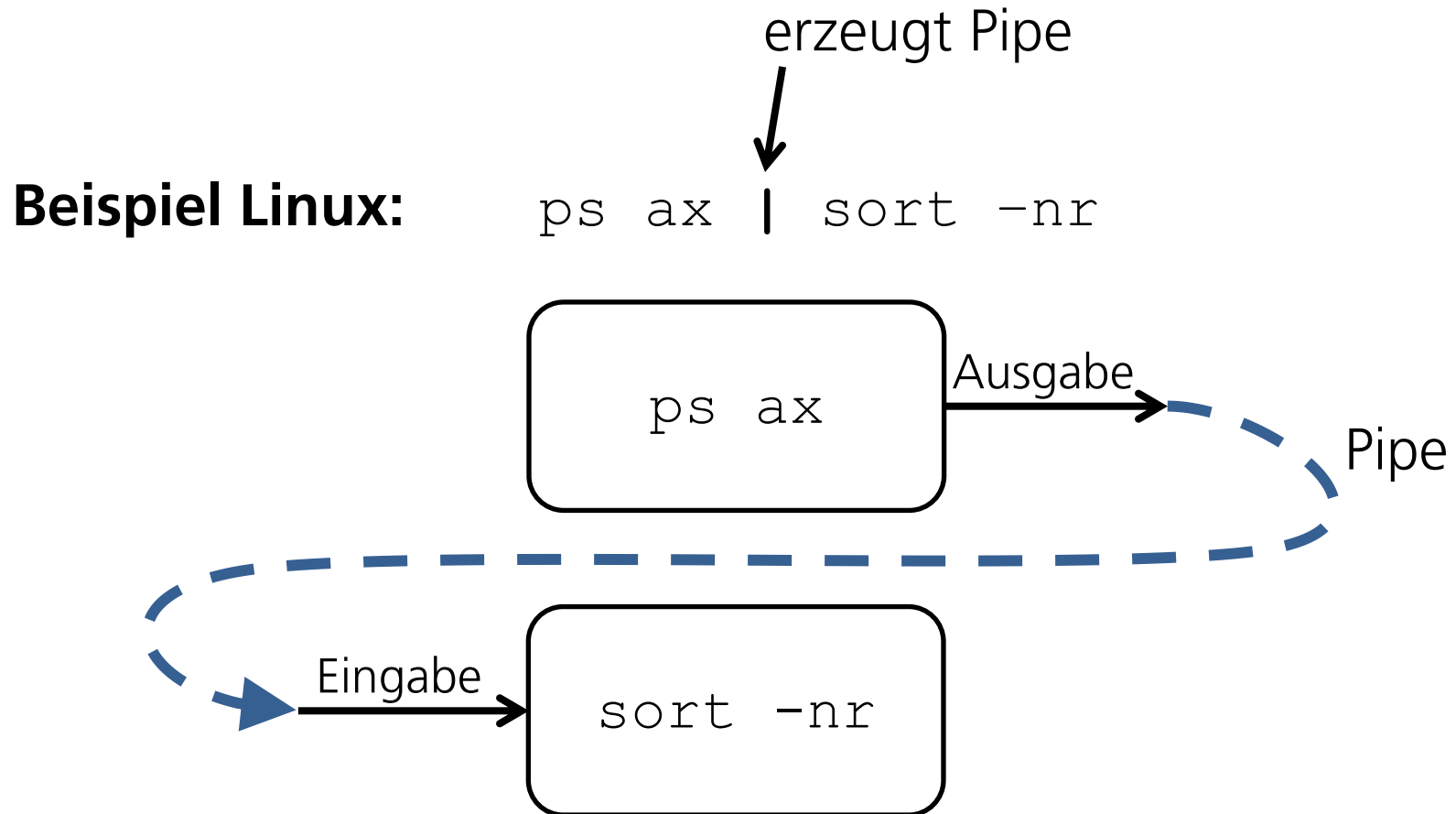
Pipes

Eine **Pipe** ist nun eine besondere Art der Umlenkung. Sie ist eine vom Betriebssystem bereitgestellte Funktionalität zur Umlenkung der Ausgabe eines Prozesses in die Eingabe eines weiteren Prozesses.



Diese Umlenkung ist ebenfalls für beide beteiligten Prozesse **transparent**!

Pipes



Listet alle laufenden Prozesse auf und sortiert diese aufsteigend nach Prozessnummer.

Nachrichtenaustausch

Alle bisher vorgestellten Mittel zur Prozesssynchronisation und -kommunikation sind auf das **lokale System** beschränkt. Will man mit Prozessen auf entfernten Systemen kommunizieren, braucht es eine Schnittstelle zum **Nachrichtenaustausch**.

Voraussetzung dafür ist das Vorhandensein eines **Kanals**, über den kommuniziert werden kann. Ein Prozess kann mehrere Kanäle gleichzeitig verwenden.

Für diesen Kanal stellt das Betriebssystem Funktionen zum **Senden** und **Empfangen** von Nachrichten bereit. Die genaue Implementierung bleibt den Prozessen verborgen.

Nachrichtenaustausch – Puffer

Der verwendete Nachrichtenkanal steht nicht zu jeder Zeit zur Verfügung. Neben anderen Gründen (bspw., weil gerade andere Stationen senden) kann dies auch daran liegen, dass der Empfänger für die Verarbeitung der Nachrichten mehr Zeit benötigt.

Aus diesem Grund werden den Sende- und Empfangsoperationen Puffer vorgeschaltet, deren Verwendung über Semaphoren synchronisiert wird.

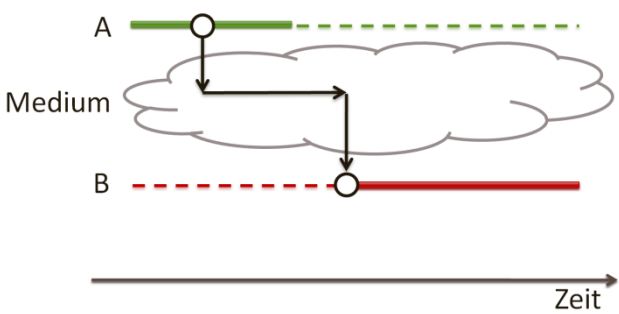
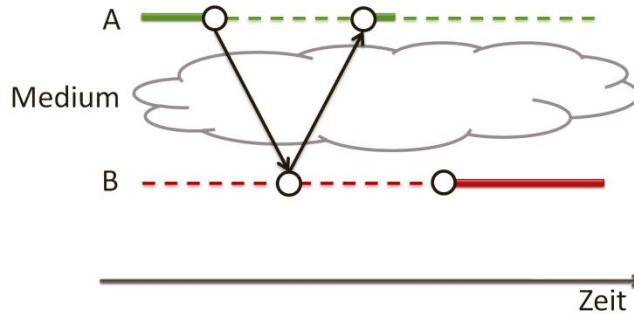
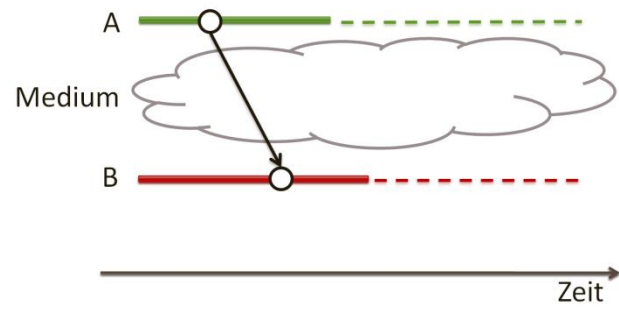
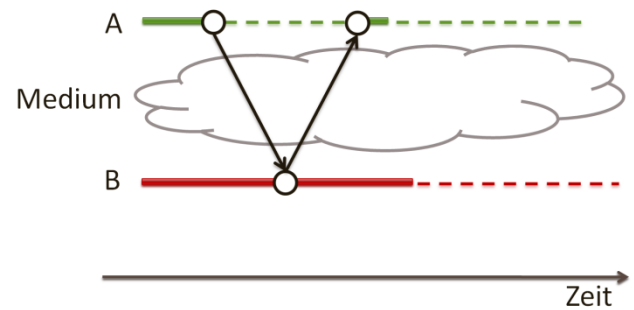
Nachrichtenaustausch – Puffer

Bei **beschränkter Kapazität** der Puffer wird

- die Sendefunktion bei **vollem** Puffer und
- die Empfangsfunktion bei **leerem** Puffer blockieren.

Bei **unbeschränkter Kapazität** der Puffer fällt die mögliche Blockade beim Sender weg.

Zur Erinnerung: Persistenz und Synchronizität

	Asynchron	Synchron
Persistent		
Transient		

Synchrone Kommunikation

Auf Ebene des **Internetprotokolls** findet Kommunikation **asynchron** statt, d.h. der Sender kann nicht sicherstellen, ob der Empfänger die Nachricht verarbeitet hat.

Synchrone Kommunikation wird hergestellt, indem der Empfänger den Empfang der Nachricht **bestätigt**:

Sender

```
sende (nachricht)
```

```
warte ();
```

```
...
```

Empfänger

```
empfange ();
```

```
bestätige (n);
```

```
...
```



Erzeuger-Verbraucher-Problem mit Nachrichten

Das Warten auf Bestätigung entspricht funktional der Synchronisation mit Semaphoren.

Zur Steigerung der Performanz kann man nun dem Sender erlauben, eine begrenzte Anzahl von Paketen **vorausend** zu senden und sich **später** bestätigen zu lassen.

Nach ebendiesem Prinzip arbeitet das **Transmission-Control-Protocol** (TCP). Es entspricht der Umsetzung des **Erzeuger-Verbraucher-Problems** auf Basis der Kommunikation mit asynchronen **Nachrichten**.

Erzeuger-Verbraucher-Problem mit Nachrichten

Nutzerprozesse

Erzeuger

```
n := erzeuge();
```

```
sende(n);
```

Verbraucher

```
n := empfange();
```

```
verarbeite(n);
```

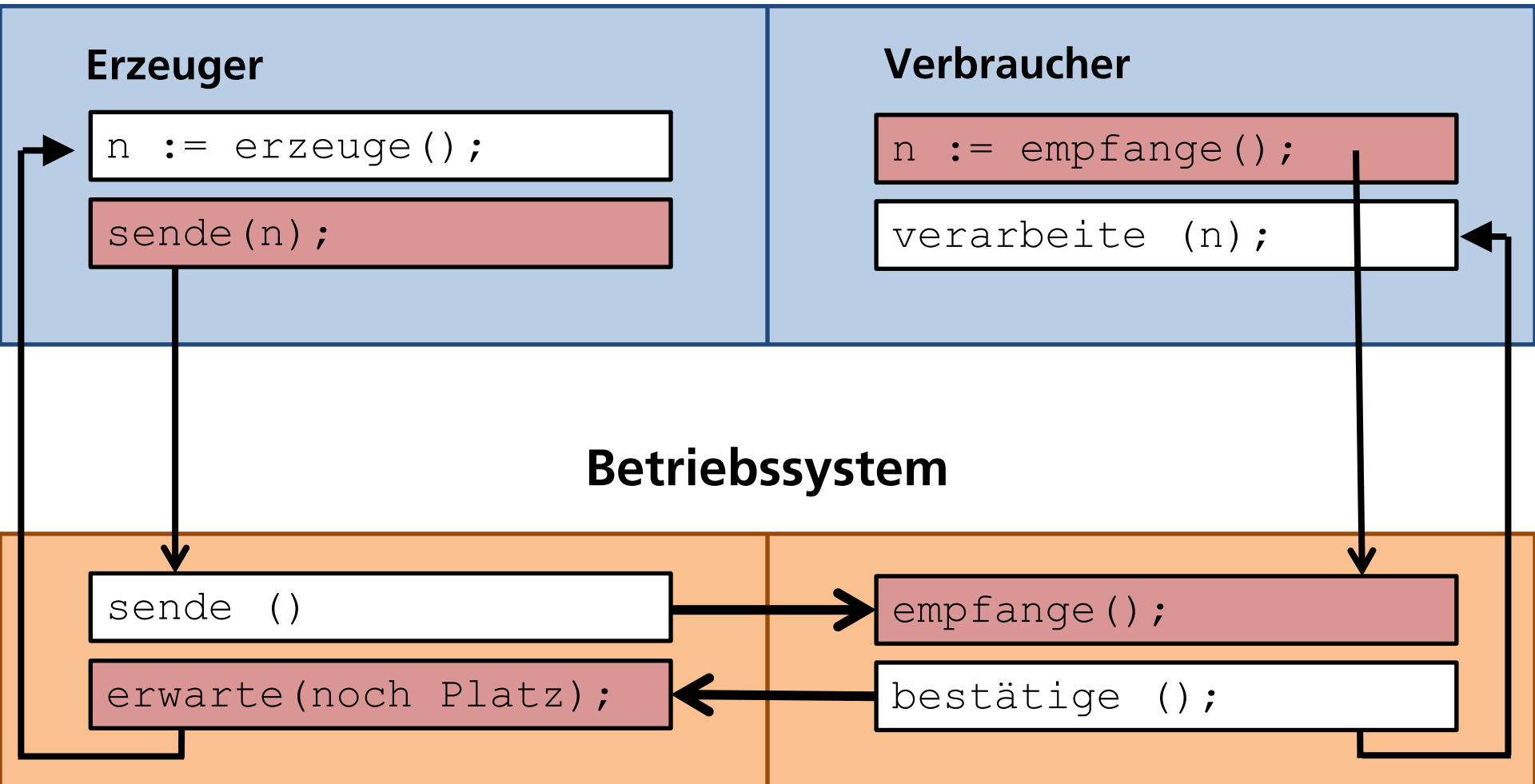
Betriebssystem

```
sende()
```

```
erwarte(noch Platz);
```

```
empfange();
```

```
bestätige();
```



Zusammenfassung

Verklemmungen können mit verschiedenen Methoden behandelt werden, die Methode der Wahl hängt vom Einsatzzweck ab.

Typische Situationen der Prozesssynchronisation werden mit standardisierten Ansätzen gelöst:

- Das Erzeuger-Verbraucher-Problem für Warteschlangen mit begrenzter Kapazität und **blockierenden** Anfragen beim Erzeuger,
- Der **schlafende Friseur** für Warteschlangen mit begrenzter Kapazität und **nicht-blockierenden** Anfragen beim Erzeuger,
- die **Readers-Writers-Probleme** für Situationen, in denen unbegrenzt viele lesende Prozesse parallelen Zugriff auf gemeinsam genutzte Ressourcen erhalten sollen, schreibende Prozesse jedoch exklusiven Zugriff.

Zusammenfassung

- Durch **Signale** können Prozesse einander Ereignisnachrichten senden und bei Bedarf individuell darauf reagieren.
- Nachrichtenaustausch mit Nutzlast ist auf einem lokalen System über **Pipes** möglich.
- Zum Nachrichtenaustausch mit entfernten Systemen stellt das Betriebssystem **geeignete Funktionen** bereit. Diese sind in der Praxis oft mit blockierenden Puffern versehen.
- Durch diese **blockierenden Puffer** ist **Synchronisation** auch mit entfernten Prozessen möglich.