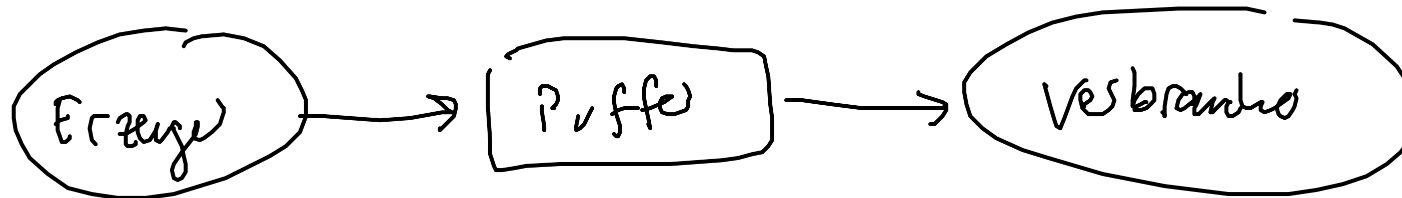
Erzeuger-Verbraucher-Problem

- Hier: Puffer der Größe 1, Erzeuger, Verbraucher



- Zwei Semaphore werden eingesetzt, um zwischen Threads "Ereignisse zu melden"
  - Man kann Semaphore auch verwenden, um Ereignisse zu produzieren und zu konsumieren
    - P = konsumiert ein Ereignis (blockiert, wenn es noch gar nicht erzeugt worden ist)
    - V = erzeugt ein Ereignis
  - Jetzt kann man sich gegenseitig Ereignisse melden
- Beispiel:
  - Semaphor leer = "Der Puffer ist leer"
  - Semaphor voll = "Der Puffer ist voll"

Beispiel

Puffer ist initial leer

- Erzeuger-Thread:

```
Loop {
  P(leer);
  // Puffer füllen
  V(voll);
}
```

- Verbraucher-Thread

```
Loop {
  P(voll);
  // Puffer leeren
  V(leer);
}

Semaphor leer = 1;
Semaphor voll = 0;
```

"Anzahl der noch nicht konsumierten Nachrichten"

Beispiel

```
Semaphor leer = 1;
Semaphor voll = 0;
```

- P1 (Erzeuger):

```
    Loop {
01    P(leer);
02    // Puffer füllen
03    V(voll);
    }
```

- P2 (Verbraucher):

```
    Loop {
04    P(voll);
05    // Puffer leeren
06    V(leer);
    }
```

Puffer	leer.z	leer.L	voll.z	voll.L	Wer	Wo	Was
leer	1	<>	0	<>	P1	1	P(leer)
leer	0	<>	0	<>	P2	4	P(voll)
leer	0	<>	-1	<P2>	P1	2	Puffer füllen
voll	0	<>	-1	<P2>	P1	3	V(voll)
voll	0	<>	0	<>	P2	5	Puffer leeren
leer	0	<>	0	<>	P2	6	V(leer)
leer	1	<>	0	<>	P1	1	P(leer)
leer	0	<>	0	<>	P1	2	Puffer füllen
voll	0	<>	0	<>	P1	3	V(voll)
voll	0	<>	1	<>	P1	1	P(leer)
voll	-1	<P1>	1	<>	...		

Bemerkungen

- Blockiert der Erzeuger bei vollem Puffer?
  - Erzeuger macht initial ein P(leer), leer wurde mit 1 initialisiert
  - Erzeuger blockiert beim zweiten P(leer), falls nicht in der Zwischenzeit ein V(leer) gemacht wurde
  - P(leer) bedeutet: gehe nur weiter, wenn Puffer leer
  - V(leer) bedeutet: sag dem Erzeuger, dass der Puffer leer ist
- Blockiert der Verbraucher bei leerem Puffer?
  - Verbraucher blockiert beim initialen P(voll), voll initialisiert mit 0
  - Erst, wenn der Erzeuger ein V(voll) gemacht hat, deblockiert Verbraucher
  - V(voll) bedeutet: sag dem Verbraucher, dass der Puffer voll ist
  - P(voll) bedeutet: gehe nur weiter, falls Puffer voll
- Welche Reihenfolge wird eingehalten?
  - Erzeuger und Verbraucher bearbeiten abwechselnd den Puffer

Erstes Leser-Schreiber-Problem

- Szenario:
  - mehrere Leser und mehrere Schreiber
  - gemeinsamer Datenbereich
  - Schreiber haben exklusiven Zugriff
  - Leser können parallel zugreifen (natürlich nur, wenn kein Schreiber zugreift)

- Implementierung mit Semaphoren
  - Semaphor W regelt den exklusiven Zugriff mittels P(W) und V(W)
  - W erlaubt durch einem Trick den parallelen Zugriff der Leser
    - Nur der erste Leser ruft P(W) auf, der letzte dann V(W)

Beispiel

```
int Readcount = 0;
Semaphor W = 1, Mutex = 1;

// Zugriff eines Leser-Threads:
P(Mutex);
Readcount++;
if (Readcount == 1) P(W);
V(Mutex);
// Daten Lesen
P(Mutex);
Readcount--;
if (Readcount == 0) V(W);
V(Mutex);

// Zugriff eines Schreiber-Threads:
P(W);
// Schreibe Daten
V(W);
```

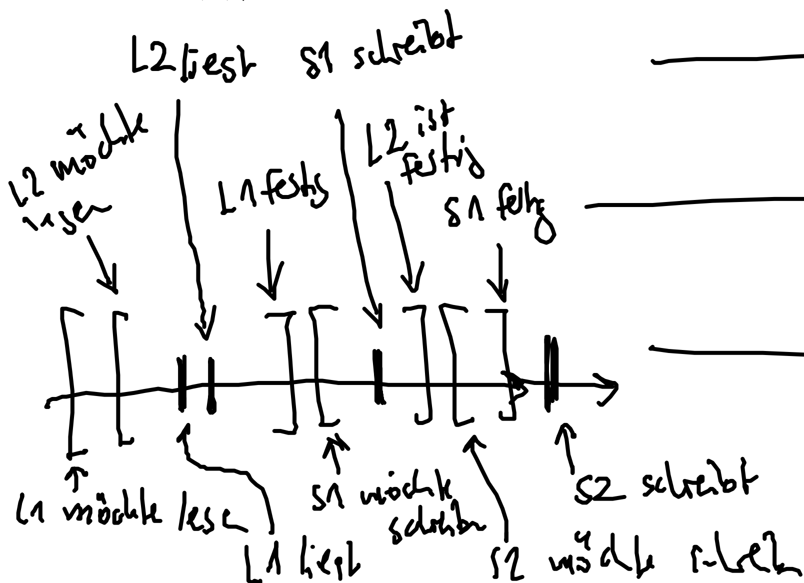
Ablaufbeispiel

```
int Readcount = 0;
Semaphor W = 1, Mutex = 1;

   // Zugriff eines Leser-Threads:
01 P(Mutex);
02 Readcount++;
03 if (Readcount == 1) P(W);
04 V(Mutex);
05 // Daten Lesen
06 P(Mutex);
07 Readcount--;
08 if (Readcount == 0) V(W);
09 V(Mutex);

   // Zugriff eines Schreiber-Threads:
10 P(W);
11 // Schreibe Daten
12 V(W);
```

Mutex.z	Mutex.L	ReadCnt	W.z	W.L	Wer	Wo	Was
1	<>	0	1	<>	L1	1	P(Mutex)
0	<>				L2	1	P(Mutex)
-1	<L2>				L1	2	Readcnt++
		1			L1	3	if.... P(W)
			0	<>	L1	4	V(Mutex)
0	<>				L2	2	Readcnt++
		2			L2	3	if....
					L2	4	V(Mutex)
1	<>				L1	5	lesen
					L2	5	lese
					L1	6	P(Mutex)
0	<>				L2	6	P(Mutex)
-1	<L2>				L1	7	Readcnt --
		1			L1	8	if....
					L1	9	V(Mutex)
0	<>				S1	10	P(W)
			-1	<S1>	L2	7	Readcnt --
		0			L2	8	if... V(W)
			0	<>	S1	11	Schreiben
					L2	9	V(Mutex)
1	<>				S2	10	P(W)
			-1	<S2>	S1	12	V(W)
			0	<>	S2	11	Schreiben
					S2	12	



Bemerkungen

- Warum schliessen sich einzelne Schreiber und die Gruppe aller Leser gegenseitig aus?
  - Um zu schreiben, muss man das Semaphor `W` passiert haben
  - Solange kein `V(W)` gemacht wurde, blockieren alle Schreiber
  - Solange noch mindestens ein Leser liest, ist das Semaphor `W` auch gesperrt, alle Schreiber blockieren am `P(W)`
- Wofür braucht man das Semaphor Mutex?
  - Ohne Mutex könnte zwischen `Readcount++` und der Überprüfung ein Thread-Wechsel stattfinden
  - Wenn zwei Leser "gleichzeitig" `Readcount` erhöhen, könnten Leser lesen, ohne ein `P(W)` gemacht zu haben
- Wie fair ist die Lösung?
  - Solange mindestens ein Leser liest, kann kein Schreiber schreiben
  - Leser können sich immer abwechseln und so alle Schreiber ausblocken (sie dürfen nur das Semaphor W nie "loslassen")

Zweites Leser-Schreiber-Problem

- Grundszenario wie erstes Leser-Schreiber-Problem
  - Jetzt sollen aber die Schreiber Priorität haben!
- Was bedeutet Priorität?
  - Zugriff eines Schreibers soll zum "frühestmöglichen" Zeitpunkt erlaubt werden
  - Wenn noch Leser lesen, aber Schreiber schreiben möchten, darf kein neuer Leser beginnen zu lesen
  - Ein Leser darf erst wieder lesen, wenn ein aktueller Schreiber aufgehört hat zu schreiben und kein Schreiber mehr schreiben will
  - Bei einer offenen Wettbewerbssituation (Datenbereich ist frei, ein Schreiber möchte schreiben, mehrere Leser möchten lesen) darf maximal ein Leser dem Schreiber zuvorkommen
- Komplexes Problem, einfache Lösung mit Semaphoren

Beispiel

```
int Readcount = 0;
int Writecount = 0;
Semaphor Mutex1 = 1,
  Mutex2 =  1,
  Mutex3 = 1,
  W = 1,
  R = 1;
```

- in blau: Code des ersten Leser-Schreiber-problems

```
// Zugriff eines Leser-Threads:
P(Mutex3);
   P(R);
      P(Mutex1);
      Readcount++;
      if (Readcount == 1) P(W);
      V(Mutex1);
   V(R);
V(Mutex3);
// lese Daten
P(Mutex1);
Readcount--;
if (Readcount == 0) V(W);
V(Mutex1)
```

```
// Zugriff eines Schreiber-Threads
P(Mutex2);
Writecount++;
if (Writecount == 1) P(R);
V(Mutex2);
P(W);
// schreibe Daten
V(W);
P(Mutex2);
Writecount--;
if (Writecount == 0) V(R);
V(Mutex2);
```

Bemerkungen

- Semaphor `W` hat dieselbe Bedeutung wie vorher
- `Readcount` hat dieselbe Bedeutung wie vorher
  - `Mutex1` schützt den exklusiven Zugriff auf `Readcount`
- `Writecount` zählt die Anzahl der bereiten Schreiber
  - `Mutex2` regelt exklusiven Zugriff auf `Writecount`
- Semaphor `R` realisiert die Priorität der Schreiber:
  - `R` definiert einen kritischen Abschnitt, um den sich sowohl Leser als auch Schreiber bewerben
  - `R` wird von Schreibern erst dann aufgegeben, wenn kein weiterer Schreiber sich beworben hat
  - Mit dem Semaphor `Mutex3` wird erreicht, dass sich alle Leser nacheinander um den Eintritt in `R` bewerben
    - So kann maximal ein Leser einem Schreiber zuvorkommen

Ablaufbeispiel

```
// Zugriff eines Leser-Threads:
P(Mutex3);
  P(R);
    P(Mutex1);
    Readcount++;
    if (Readcount == 1) P(W);
    V(Mutex1);
  V(R);
V(Mutex3);
// lese Daten
P(Mutex1);
Readcount--;
if (Readcount == 0) V(W);
V(Mutex1)
```

```
// Zugriff eines Schreiber-Threads
P(Mutex2);
Writecount++;
if (writecount == 1) P(R);
V(Mutex2);
P(W);
// schreibe Daten
V(W);
P(Mutex2);
Writecount--;
if (Writecount == 0) V(R);
V(Mutex2);
```

Readcount	W.z	W.L	R.z	R.L	Mutex3.z	Mutex3.L	Kommentar
0	1	<>	1	<>	1	<>	L1 beginnt zu lesen
1	0	<>	1	<>	1	<>	L2 beginnt zu lesen
2	0	<>	1	<>	1	<>	L3, L4 wollen lesen, S1 will schreiben
							L3 holt sich mutex3, S1 holt sich R
2	-1	<S1>	-1	<L3>	-1	<L4>	L1 beendet lesen
1	-1	<S1>	-1	<L3>	-1	<L4>	L2 beendet lesen ⇒ V(W)
0	0	<>	-1	<L3>	-1	<L4>	S1 beendet schreiben
0	1	<>	0	<>	-1	<L4>	L3 beginnt zu lesen ⇒ V(Mutex3)
1	0	<>	1	<>	0	<>	L4 beginnt zu lesen
2	0	<>	1	<>	1	<>	⋮

Erläuterung

- Leser1 beginnt zu lesen, anschliessend ist Readcount = 1, Semaphor W ist belegt
- Leser2 möchte lesen, Readcount = 2, geht direkt in den kritischen Abschnitt
- Leser3, Leser4 und Schreiber1 treten in direkte Wettbewerbssituation: alle drei starten parallel ihr Eintrittsprotokoll
  - Leser3 und Leser4 streiten sich um Mutex3: Nur einer (z.B. Leser3) kommt durch, Leser4 blockiert an P(Mutex3)
  - Leser3 und Schreiber konkurrieren um R: Nur einer (z.B. Schreiber) kommt durch, Leser3 blockiert an P(R)
    - Schlimmstenfalls hätte Schreiber nur einen Leser vorlassen müssen
  - Ergebnis: Leser4 blockiert an P(Mutex3), Leser3 blockiert an P(R), Schreiber blockiert an P(W)
- kein neuer Leser kommt in den kritischen Abschnitt
- Wenn Leser1 und Leser2 den kritischen Abschnitt verlassen, deblockiert Schreiber und geht in seinen kritischen Abschnitt

Bedingte kritische Abschnitte

- Eintritt in den kritischen Abschnitt hängt manchmal von einem Prädikat ab
  - Prädikat kann über den im kritischen Abschnitt geschützten Daten definiert sein und muss deswegen auch geschützt werden
- Beispiel: DVD-Laufwerke, die auch defekt sein können
  - Eintritt nur, falls es noch ein freies DVD-Laufwerk gibt, was nicht defekt ist
- Naive Implementierung:

```
Semaphore Mutex = 1;
P(Mutex);
while (!Prädikat) {
  V(Mutex); NOP; P(Mutex);
}
// Datenzugriff
V(Mutex);
```

Bessere Lösung

- Naive Lösung implementiert wieder aktives Warten!
  - In diesem Fall kann es bei ungünstiger Prozessorzuteilung zu einem Aushungern der Threads kommen
- Idee der besseren Lösung:
  - Nur wenn die Daten modifiziert wurden, braucht man das Prädikat neu zu überprüfen
  - Blockiere solange an zweitem Semaphor `Condsem`, bis die Daten modifiziert wurden
  - Mit Zähler `Waitcount` wird gezählt, wieviele Threads die Bedingung prüfen wollen
  - Bei Modifikation der Daten, `Waitcount`-Mal ein Ereignis auf `Condsem` generieren

Beispiel

```
Semaphore Mutex = 1, Condsem = 0;
int Waitcount = 0;
...
P(Mutex);
while (!Prädikat) {
  Waitcount++;
  V(Mutex);
  P(Condsem);
  P(Mutex);
}
// Datenzugriff
while (Waitcount > 0) {
  Waitcount--;
  V(Condsem);
}
V(Mutex);
```

Zusammenfassung Semaphore

- Semaphore: Üblicherweise implementiert auf Ebene des Dispatchers
  - Bei UL-Threads kann man Semaphore auch auf Anwendungsebene realisieren (mehr siehe Übung)
  - Für Echtzeitverarbeitung können die Semaphor-Listen prioritätsgesteuert werden
- Semaphore sind eine sehr mächtige Abstraktion
  - Man kann mit ihnen schnell und einfach Lösungen zu Synchronisationsproblemen schreiben
  - Komplexe Anforderungen resultieren oft in komplexen, fehleranfälligen Lösungen
  - Semaphore sind für die Betriebssystemprogrammierung gut, schlecht für die Programmierung auf Anwendungsebene
- Gesucht: Sprachkonzept für leichter verständliche Lösungen auf Ebene einer Programmiersprache

Übersicht

- Einführung: Kritische Abschnitte
- Hardwaregestützte Mechanismen
- Betriebssystemgestützter Mechanismus: Semaphore
- **Sprachgestützter Mechanismus: Monitore**
  - **Programmiersprachliche Realisierung**
  - **Implementierung mit Semaphoren**
- Realisierungsbeispiele

Monitorkonzept

- Schon in den 1970er Jahren wurde klar, dass für die Software-Entwicklung noch abstraktere Synchronisationsformen notwendig waren
- Das Monitorkonzept fand dabei die meiste Beachtung
  - Zeitgleich erfunden von Brinch Hansen und Hoare
- Monitore kapseln kritische Daten und kritische Abschnitte in eine programmiersprachliche Einheit
  - Monitore kann man sich als Module aus modulorientierten Sprachen oder Klassen in objektorientierten Sprachen denken
  - Monitore fügen aber eine explizite Synchronisationssemantik hinzu
- Monitor = kritischer Abschnitt
  - Es ist immer nur ein Prozess gleichzeitig "im Monitor" (d.h. führt eine Monitorprozedur aus)

Sprachliche Realisierung

```
MONITOR Monitorname (Parameter)
  Datendeklaration // gemeinsame Daten
  ENTRY Funktionsname1 (Parameter) { Prozedurkörper
  }
  ENTRY Funktionsname2 (Parameter) { Prozedurkörper
  }
  ...
  ENTRY FunktionsnameN (Parameter) { Prozedurkörper
  }
  INIT { Initialisierungscode }
END
```

- Initialisierung eines Monitors: einmaliges Aufrufen von
  - `Monitorname(aktuelle Parameter)`
- Aufruf einer Monitorprozedur:
  - `Monitorname.Funktionsname(aktuelle Parameter)`

Vorteile von Monitoren

- Synchronisation
  - Die Programmiersprache garantiert, dass auch bei mehreren parallelen Threads immer nur ein Thread gleichzeitig eine Monitorprozedur ausführt
  - Monitorprozeduren sind also logisch atomar
- Kritische Daten werden explizit in der Programmstruktur sichtbar gemacht
  - Die Zugriffsoperationen definieren die zulässigen Manipulationen der Monitordaten
  - Ein Umgehen des Monitors ist nicht möglich
- Kapselung kritischer Daten
  - Wie bei Modulen oder Klassen realisieren Monitore das Konzept des *information hiding*
    - Zugriffsfunktionen verbergen die interne Struktur
  - Erhöht die Wartbarkeit des Programmcodes

Beispiel: exklusive Ressourcen

- Jede exklusive benutzbare Ressource wird durch einen Monitor repräsentiert
  - Prozeduren definieren die Zugriffsfunktionen
- Beispiel: Monitor Disc zur Synchronisation von Plattenzugriffen

```
MONITOR Disc
  ENTRY Read (Festplattenaddresse, Speicheradresse) {
    // Lesevorgang durchführen
  }
  ENTRY Write (Speicheradresse, Festplattenadresse) {
    // Schreibvorgang durchführen
  }
  INIT {
    // Gerät initialisieren
  }
END
```