Leser-Schreiber-Realisierung mit Semaphoren

Reader:

```
...
        down(semWriter);
        down(semCounter);
        rcounter++;
        up(semCounter);
        up(semWriter);

        read();

        down(semCounter);
        rcounter--;
        up(semCounter);
...
```

Writer:

```
...
        down(semWriter);

        while(true)
        {
                down(semCounter);
                if(rcounter==0)
                        break;
                up(semCounter);
        }
        up(semCounter);

        write();

        up(semWriter);
...
```

Problem:
Busy Waiting –
siehe spätere Lösung

Verbessertes Konzept: Monitore

- Ein Nachteil von Semaphoren ist die Notwendigkeit zur expliziten Anforderung P und Freigabe V des kritischen Bereiches durch den Programmierer
- Vergisst der Entwickler z.B. die Freigabe V des Semaphors nach dem Durchlaufen des kritischen Abschnitts, dann kann es schnell zu einer Verklemmung kommen; solche Fehler sind sehr schwer zu finden!
- Zum einfacheren und damit weniger fehlerträchtigen Umgang mit kritischen Bereichen wurde deshalb das Konzept der *Monitore* (Hoare 1974, Brinch Hansen 1975) entwickelt:
  - Ein **Monitor** ist eine Einheit von Daten und Prozeduren auf diesen Daten, auf die zu jeden Zeitpunkt nur maximal ein Prozess zugreifen kann.
  - Wollen mehrere Prozesse gleichzeitig auf einen Monitor zugreifen, so werden alle Prozesse bis auf einen Prozess in eine Warteschlange eingereiht und blockiert.
  - Verlässt ein Prozess den Monitor, so wird ein Prozess aus der Warteschlange entnommen und dieser kann auf die Funktionen und Daten des Monitors zugreifen.
  - Die Signalisierung ist innerhalb des Monitors festgelegt, der Programmierer muss sie nicht selbstständig implementieren.

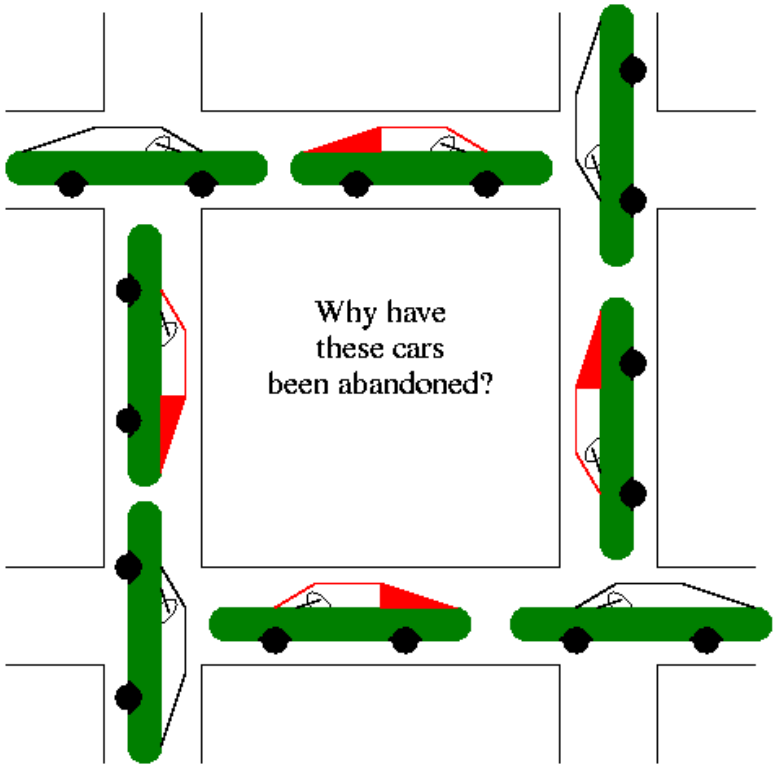
Beispiel: Monitore in Java

- In Java werden Monitore durch `synchronized`-Methoden implementiert. Zu jedem Zeitpunkt darf nur ein Prozess sich **aktiv** in einer dieser Methoden befinden.
- **Anmerkung**: normalerweise werden höhere Konstrukte wie Monitore durch einfachere Konstrukte wie den Semaphor implementiert.
  Siehe auch die Realisierung von Semaphoren durch das einfachere Konzept TSL-Befehl.
- In Java kann man das Monitorkonzept allerdings auch nutzen um selber Semaphore zu implementieren (siehe nebenstehenden Code).
- `wait()` und `notify()` sind zu jedem Objekt in Java definierte Methoden.

```
public class Semaphore {
        private int value;

        public Semaphore (int initial) {
                value = initial;
        }

        synchronized public void up() {
                value++;
                if(value==1) notify();
        }

        synchronized public void down() {
                while(value==0) wait();
                value- -;
        }
}
```

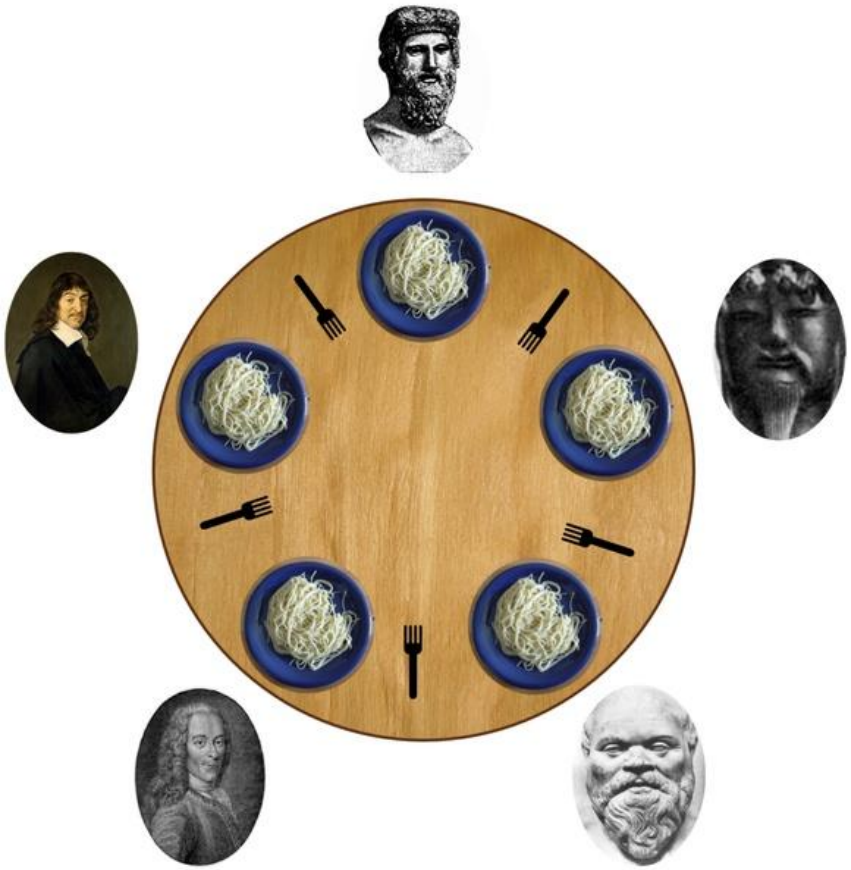
Bemerkung zu Verklemmungen (Deadlocks)

- Auch bei der korrekten Verwendung von Semaphoren und Monitoren kann es zu Deadlocks kommen, siehe Beispiel auf der folgenden Seite.
- Coffman, Elphick und Shoshani haben 1971 die vier konjunktiv notwendigen Voraussetzungen für einen Deadlock formuliert:
  1. Wechselseitiger Ausschluss: Es gibt eine Menge von exklusiven Ressourcen R_{exkl}, die entweder frei sind oder genau einem Prozess zugeordnet sind.
  2. Hold-and-wait-Bedingung: Prozesse, die bereits im Besitz von Ressourcen aus R_{exkl} sind, fordern weitere Ressourcen aus R_{exkl} an.
  3. Ununterbrechbarkeit: Die Ressourcen R_{exkl} können einem Prozess nicht entzogen werden, sobald er sie belegt. Sie müssen durch den Prozess explizit freigegeben werden.
  4. Zyklische Wartebedingung: Es muss eine zyklische Kette von Prozessen geben, die jeweils auf Ressourcen warten, die dem nächsten Prozess in der Kette gehören.
- Umgekehrt (und positiv) formuliert: ist eine der Bedingungen nicht erfüllt, so sind Verklemmungen ausgeschlossen.



Klassisches Beispiel: Speisende Philosophen

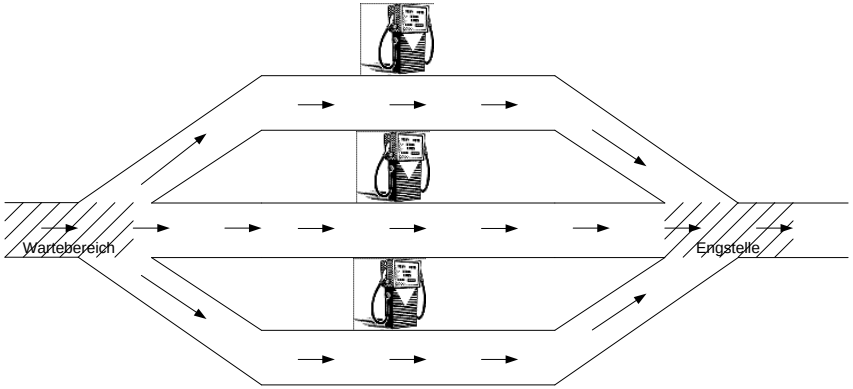
- Klassisches Beispiel aus der Informatik für Verklemmungen: "Dining Philosophers" (speisende Philosophen, Dijkstra 1971, Hoare 1971)
- 5 Philosophen (Prozesse) sitzen an einem Tisch. Vor ihnen steht jeweils ein Teller mit Essen. Zum Essen benötigen sie zwei Gabeln (Betriebsmittel), insgesamt sind aber nur 5 Gabeln verfügbar.
- Die Philosophen denken und diskutieren. Ist einer hungrig, so greift er zunächst zur linken und dann zur rechten Gabel. Ist eine Gabel nicht an ihrem Platz, so wartet er bis die Gabel wieder verfügbar ist (ohne eine evtl. in der Hand befindliche Gabel zurückzulegen). Nach dem Essen legt er die Gabeln zurück.
- Problem: sind alle Philosophen gleichzeitig hungrig, so nehmen sie alle ihre linke Gabel und gleichzeitig ihrem Nachbarn die rechte Gabel weg. Alle Philosophen warten auf die rechte Gabel und es entsteht eine Verklemmung (deadlock).
- Gibt ein Philosoph seine Gabel nicht mehr zurück, so stirbt der entsprechende Nachbar den **Hungertod (starvation)**.



Klausur WS06/07 – Nebenläufigkeit (15 Punkte = 15min)

Prozess: tankendes Auto

```
fahreInWartebereich();

fahreAnZapfsaeule();

tanke();

bezahle();

fahreInEngstelle2();

verlasseEngstelle2();
```



a) Geben Sie die notwendigen Semaphore (mitsamt Initialisierung) an, um das gegebene Problem zu lösen.
Beispiel: semAuto(1) würde bedeuten, Sie verwenden einen Semaphor semAuto, der mit 1 initialisiert ist.

b) Ergänzen Sie den folgenden Autoprozess mit passenden up() und down()-Methoden, um Kollisionen zu vermeiden. Achten Sie darauf, dass es zu keiner Verklemmung kommt. **Anmerkung**: Es muss nicht an jeder freien Stelle Code eingefügt werden.
Beispiel: 1: down(semaAuto); up(semaAuto); bedeutet das Einfügen der beiden Operationen in Zeile 1.

Klausur WS06/07 - Nebenläufigkeit

c) Aufgrund einer Baustelle ist die Ausfahrt blockiert (siehe Abbildung), so dass die Wartebereich sowohl zur Einfahrt, als auch zur Ausfahrt genutzt werden muss. Ergeben sich notwendige Änderungen im Vergleich zur Lösung der Aufgabe b) und wenn ja welche?

Prozess: tankendes Auto

```
fahreInWartebereich();

fahreAnZapfsaeule();

tanke();

bezahle();

fahreInEngstelle2();

verlasseEngstelle2();
```