

Synchronisation

Bei der **Synchronisation** geht es um Mechanismen zur Vermeidung von Problemen, die bei der **nebenläufigen Ausführung von Prozessen** oder Threads in Verbindung mit **gemeinsam genutzten Betriebsmitteln** entstehen können.

Nebenläufigkeit ist quasi-parallele Ausführung von Befehlen mehrerer Prozesse oder Threads mit Kontextwechsel auf CPU.

Beim **Race Conditions** nutzen zwei oder mehr Prozesse ein oder mehrere Betriebsmittel gemeinsam, wobei das Ergebnis der Ausführung von der zeitlichen Reihenfolge der Zugriffe der Prozesse oder Threads auf das Betriebsmittel abhängt.

kritischer Abschnitt sind Programmteile, die während Ausführung auf der CPU **nicht** durch kritische Abschnitte anderer Prozesse oder Threads unterbrochen werden dürfen, sofern Prozesse **auf gemeinsam genutzte Betriebsmittel zugreifen**.

Aktives Warten - Prozesse und Threads synchronisieren

Unter **aktivem Warten** versteht man ständige Abfragen eines Sperrkennzeichens am Eingang eines kritischen Abschnitts.

- dafür wird das **Polling** genutzt, um ständig *Abfragen* durchzuführen
- um einen Prozess oder einen Thread zu sperren wird das genannte **Sperrkennzeichen**, was sich als *Sperrzeichen* äußert

```
public class Beispiel_Aktives_Warten {

    static int counter = 0;        // gemeinsam genutztes Betriebsmittel
    static int lock = 0;           // Sperrvariable

    public static class Thread_A extends Thread {
        public void run() {
            do_something();        // unkritisch
            count_from_10();       // kritisch !!!
        }
    }
}
```

```

do_something_else();    // unkritisch
}

private void do_something() {
    // unkritischer Abschnitt
    System.out.println("Thread_A: unkritisch");
}

private void count_from_10() {
    // Vorsicht: kritischer Abschnitt!
    while (lock == 1);    // wenn lock auf 1 ist, dann ist das Thread in der While-Schleife
    // gefangen
    lock = 1; // wenn lock 0 war wird er hier auf 1 gesetzt und die Methode durchlaufen
    counter = 10;
    counter++;
    counter++;
    System.out.println("A-Counter: " + counter);
    lock = 0; // die methode ist durchlaufen, auch wenn zwischen durch ein anderer
    // Thread dran war, ab hier kann ein anderer Thread durchlaufen werden
    private void do_something_else() {
        // unkritischer Abschnitt
        System.out.println("Thread_A: wieder unkritisch");
    }
}

public static class Thread_B extends Thread {
    public void run() {
        System.out.println("Thread_B ist gestartet.");
        while (lock == 1);    // Semikolon beachten!
        lock = 1;
        counter = 20;
        counter++;
        counter++;
        System.out.println("B-Counter: " + counter);
        lock = 0;
    }
}

public static void main(String[] args) {
    Thread a = new Thread_A();
    Thread b = new Thread_B();
    a.start();
}

```

```

p.start();
}

}

```

- Kontextwechsel im **ungünstigsten Moment** kann dazu führen, dass zwei oder mehr Prozesse oder Threads sich gleichzeitig in ihrem **kritischen Abschnitt befinden**.
 - es gibt Mechanismus, welcher eine Sperrvariable innerhalb einer **atomaren Aktion** abfragen und setzen kann
 - eine solche Aktion ist *atomar*, ausgeführt werden müssen und (logisch) *nicht unterbrochen* werden dürfen
- Eine Möglichkeit bietet dafür der **TSL-Assemblerbefehl**:
 - CPU sorgt bei der Ausführung des TSL-Befehls dafür, dass zwei Dinge innerhalb einer Aktion (*atomar!*) passieren, weder ein Kontextwechsel, noch ein Interrupt können den TSL-Befehl unterbrechen
 - **Kopiere** den aktuellen **Wert der lock-Variablen** aus *Speicherzelle 25* in das *Register R1*
 - **Schreibe** in *Speicherzelle 25* den Wert 1, setze also **lock = 1**.
 - **EQUAL-Befehl** vergleicht **Wert aus Register 1** mit der **Zahl 0**.
 - *Abhängig vom Ergebnis muss Prozess entweder in Schleife warten oder darf kritischen Bereich ausführen.*

```

10: TSL 25          ; Hier passieren zwei Dinge als atomare
Aktion:
[]                ; --> Wert aus Speicherzelle 25 in Akkumulator
kopieren
                  ; --> Zahl 1 in Speicherzelle 25 schreiben (setze lock=1)
11: EQUAL #0       ; Prüfe: Ist ACC == 0? (Eigentlich: ist lock == 0?)
12: JUMP 10        ; Prüfung ergab FALSE
13: ...           ; Prüfung ergab TRUE

```

Revision #1

Created 24 September 2022 16:23:00 by Merith Holtmann

Updated 2 October 2022 19:22:32 by Merith Holtmann