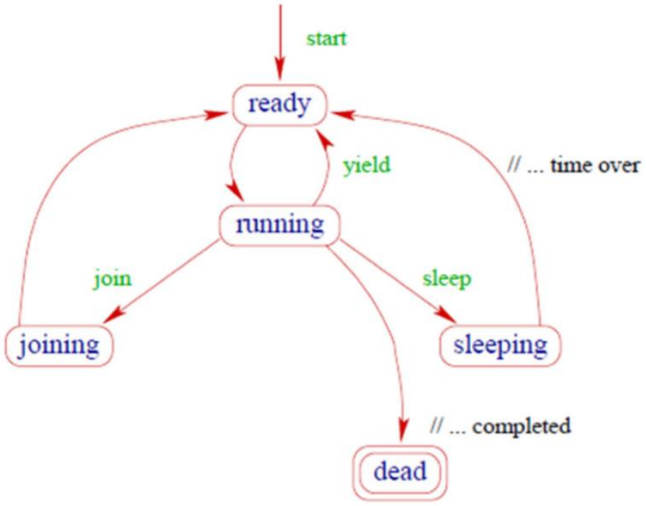
Aufgabe 9 – Threads [12]

a) Teilaufgaben: Antworten u. Grafiken größtenteils den Vorlesungsfolien von A. Brüggemann-Klein entnommen.

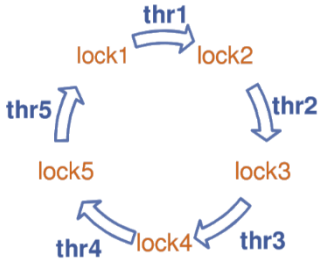
a. Erläutern Sie die Zustände `ready`, `running`, `sleeping`, `dead`, in denen sich ein Thread befinden kann.

A.: Den Zustand „ready“ erhält ein Thread nachdem er erstellt und gestartet wurde; dieser Status zeigt an, dass der Thread bereit ist für die Ausführung („rechenbereit“). Der Scheduler/Dispatcher kann einem Thread Rechenzeit zusprechen (und somit in den Zustand „running“ versetzen) oder wieder entziehen (und zurück in den Zustand „ready“ versetzen). Im Zustand „running“ arbeitet der Thread seine run()-Methode ab, bis er schließlich terminiert (und in den Zustand „dead“ versetzt wird). Im Zustand „dead“ wurde der Thread beendet. Über die Methode Tread.sleep(long) kann ein Thread in den Zustand „sleeping“ gebracht werden, in dem er entsprechend lange wartet und nach Ablauf der Zeit wieder in den Zustand „ready“ zurückkehrt.



b. Was ist ein Deadlock?

A.: „Situation von einer Gruppe von Threads, die wechselseitig auf Locks warten, die ein anderer Thread der Gruppe besitzt (thr_i hat lock_i und wartet auf lock_i+1)“



b) Diese Lösung ist getestet und korrekt – ob sie aber „mustergültig“ ist, das sei dahingestellt.

```
public static int sum(int[] input, int numThreads) {
    // Eigene Ergebnis-Klasse (statt int; um final zu ermöglichen)
    class Result {
        private int result = 0; // Gesamtergebnis aller Threads

        // Synchronisierte Methode zum Aufaddieren auf das Ergebnis.
        synchronized public void add(int value) {
            result += value;
        }

        // Gibt das Ergebnis als Integer zurück.
        public int get() {
            return result;
        }
    }

    final Result result = new Result(); // Ergebnis des Summierens
    final Thread[] t = new Thread[numThreads]; // Summier-Threads
    int segmentStd = input.length/numThreads; // Länge eines Abschnitts

    // Erstelle alle Threads
    for (int i = 0; i < numThreads; i++) {
        final int from = i*segmentStd; // Startindex zum Addieren
        final int to; // Endindex zum Addieren (exkl.)
        if (i == numThreads-1) // letzter Thread
            to = input.length;
        else // normaler Thread
            to = (i+1)*segmentStd;

        t[i] = new Thread(new Runnable() {
            @Override
            public void run() {
                // Addiere alle Zahlen auf das Ergebnis
                for (int j = from; j < to; j++) {
                    result.add(input[j]);
                }
            }
        });

        t[i].start();
    }

    // Joine alle Threads (daher ist das lokale t-Array nötig)
    for (int i = 0; i < numThreads; i++) {
        try {
            t[i].join();
        } catch (InterruptedException e) {

        }
    }

    return result.get(); // Ergebnis zurückgeben
}
```

Aufgabe 7 [9 Punkte] **Threads**

Gegeben sei der folgende Code:

```
1  public class Konflikt extends Thread {
2
3      static Integer number = 1;
4      int id;
5      int tmp;
6
7      public Konflikt(int id) {
8          this.id = id;
9          this.start();
10     }
11
12     public static void main(String[] args) {
13         for (int i = 1; i <= 3; i++) {
14             new Konflikt(i);
15         }
16     }
17
18     public void run() {
19         read();
20         compute();
21         write();
22     }
23
24     public void read() {
25         synchronized (number) {
26             tmp = number;
27         }
28     }
29
30     public void write() {
31         synchronized (number) {
32             number = tmp;
33         }
34     }
35
36     public void compute() {
37         System.out.println("" + tmp);
38         if (id == 1) {
39             tmp = tmp + 2;
40         } else if (id == 2) {
41             tmp = tmp * 3;
42         } else {
43             tmp = tmp * 2;
44         }
45     }
46
47 }
```

Geben Sie alle möglichen Ausgaben an, die durch Zeile 37 vom jeweiligen Thread mit der gegebenen `id` erzeugt werden können:

Konflikt.id==1: 1, 2, 3, 6

Konflikt.id==2: 1, 2, 3, 4, 6

Konflikt.id==3: 1, 3, 5, 9

Wir prüfen welche Abfolgen von Lese-/Schreiben möglich sind. Ein Thread muss die run()-Methode fertig abgearbeitet haben bevor ein anderer liest. Ist das nicht der Fall, so wird sein Ergebnis nicht in number zurückgeschrieben und der nächste Thread benutzt zu tmp einfach den alten Wert. Beispiel-haft für Konflikt.id==1 ergibt sich:

Erst T1, Rest egal (1); erst T2 (schreibt 3), dann T1 (ist dann 3 aus); erst T3, dann T1 (2); erst T2, dann T3, dann T1 (6); erst T3, dann T2, dann T1 (5).

Aufgabe 8 – Threads [13]

```
class SumThread extends Thread {
    int result;
    Tree t;

    SumThread(Tree t) {
        this.t = t;
    }

    public void run() {
        SumThread[] teilSumme = new SumThread[t.children.length];

        // Erstelle und starte Teil-Threads zur Berechnung der
        // Summe der einzelner Kindbäume.
        for (int i = 0; i < teilSumme.length; i++) {
            teilSumme[i] = new SumThread(t.children[i]);
            teilSumme[i].start();
        }

        result = t.data; // Eigenen Wert zum Ergebnis addieren.

        // Joine die Threads (d. h. warte bis deren Ergebnis
        // feststeht) und addiere die Ergebnisse auf.
        for (int i = 0; i < teilSumme.length; i++) {
            try {
                teilSumme[i].join();
            } catch (InterruptedException e) {
                // -> Problem (lassen wir mal weg)
            }

            result += teilSumme[i].result;
        }
    }

    public static void main(String args[]) {
       // ...
    }
}
```

S. 26 – Data race

3 Lösungen online, nur irgendwelche Lösungen hier aufzuführen soll nicht der Sinn sein, das kann man selbst testen, indem man die Java-Dateien herunterlädt und mehrfach ausführt.

Warum kann es überhaupt zu verschiedenen Ergebnissen kommen? Die Threads arbeiten doch nicht auf einer statischen (gemeinsamen) Variable, sondern besitzt jeder Thread sein eigenes privates Attribut `int[] data`*?*
- Das Problem ist, dass beim Erstellen eines neuen Threads hier als Parameter eine *Referenz* auf das Array übergeben wird, nicht aber das Array kopiert wird, wodurch beide Threads auf dasselbe Array (auf dem Heap) zugreifen/schreiben, sodass Asynchronität vorliegt.

Intern (Methode `run()`) besitzen beide Threads eine Variable `temp`. Diese hat wiederum jeder Thread für sich und wird durch andere Threads nicht beeinflusst, da diese einen konkreten *Wert* enthält und keine Referenz (>> einfacher Datentyp!).

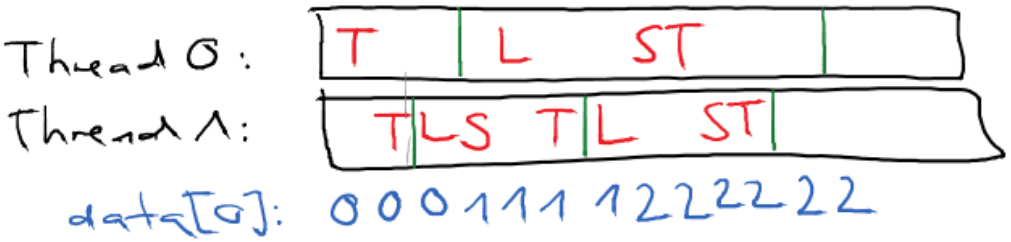
Die Threads werden beliebig „verschachtelt“ (⇒ *Interleaving*) ausgeführt, sodass es prinzipiell auch möglich wäre, dass ein Thread überhaupt nicht in die `while`-Schleife kommt, da der andere Thread bereits zweimal inkrementiert hat (praktisch sehr unwahrscheinlich). Man kann sich nun anhand einer Timeline beliebige Verläufe skizzieren; zu Beachten ist nur, dass die Reihenfolge der Ausführung pro Thread sequentiell bleibt (also z. B nicht geschrieben wird, bevor man liest).

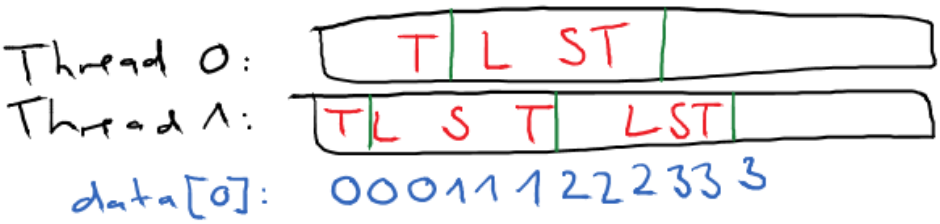
Wie gehe ich am besten vor?
Auf der nächsten Seite ist ein Beispiel, wie man sich dieses *Interleaving* und das damit verbundene (da nicht synchronisiert) *Data Race*, veranschaulichen kann.

Testen (1) → int temp = data[0]

while { Lesen → temp = data[0]
Schreiben → data[0] = temp+1
Testen (2) → temp = data[0]

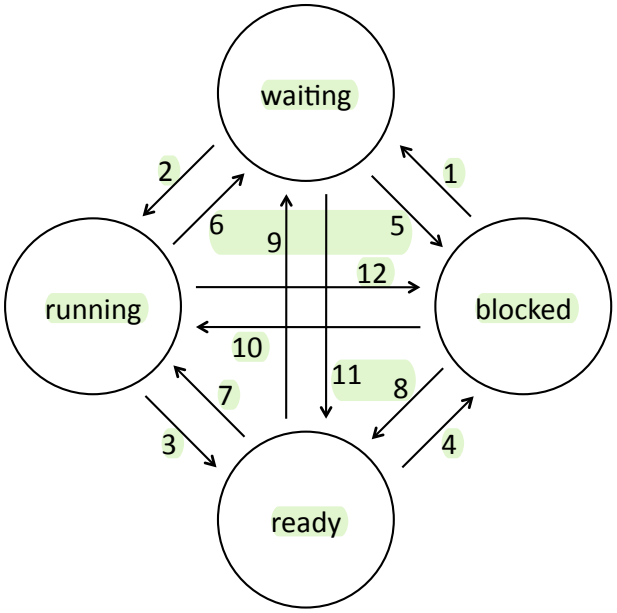
Die while-Bedingung (temp < 2) wird beliebig zwischen T und L abgefragt und bezieht sich auf den zuletzt getesteten Wert (kann von data[0] abweichen (→ data[0] bei T)





11.) Threads (11 Punkte)

(a) Im folgenden Diagramm sind die aus der Vorlesung bekannten Thread-Zustände aufgezeichnet.



Nachfolgend finden Sie Beispielereignisse, welche mit Pfeilen in der Abbildung korrespondieren.

a) Über einen Semaphor ist einem Thread der Eintritt in einen kritischen Abschnitt verweigert worden.

b) Eine vom Scheduler zugeordnete Zeitscheibe für einen laufenden Thread ist abgelaufen.

c) Ein Thread wurde vom Scheduler zur weiteren Ausführung ausgewählt.

d) Eine längere Ein-/Ausgabe-Operation wurde beendet.

e) Ein Thread möchte eine `synchronized`-Methode aufrufen, die bereits von einem anderen Thread abgearbeitet wird.

f) Über einen Semaphor wird nach Austritt eines anderen Threads aus einem kritischen Bereich dem zu betrachtenden Thread der Eintritt in diesen Bereich zugesprochen.

g) Starten einer längeren Ein-/Ausgabe-Operation.

Tragen Sie in der nachfolgenden Tabelle zu jedem Ereignis die Nummer der dazugehörigen Pfeils ein.

a)	b)	c)	d)	e)	f)	g)
6	3	7	8	12	11/2	12

Anmerkung zu f): Laut Skript wird durch `obj.notify()` bzw. `obj.notifyAll()` vom Zustand „Waiting“ in den Zustand „Ready“ gewechselt mit der Auflage, dass das Monitor-Lock von `obj` implizit neu akquiriert werden muss. Da dies für den Programmierer transparent geschieht, gilt an dieser Stelle ausser Antwort 11 auch die Antwort 2.

(b) Gegeben seien die folgenden Klassendeklarationen zur Realisierung eines „Producer-Consumer-Szenarios“ mit Threads in Java. Mit Hilfe der Klasse `MessageContainer` tauschen Instanzen von `Producer` und `Consumer` Nachrichten in Form von Ganzzahlen aus.

```
class MessageContainer {

  private static int msg;
  private static Object lock = new Object();
  private static boolean newMsgArrived = false;



  public static int get() throws InterruptedException {
    synchronized (lock) {

      while (!newMsgArrived) {
        lock.wait();
      }

      newMsgArrived = false;

      lock.notifyAll();

      return msg;









  }
}

public static void put(int newMsg) throws InterruptedException {
  synchronized (lock) {

    while (newMsgArrived) {
      lock.wait();
    }

    newMsgArrived = true;

    msg = newMsg;

    lock.notifyAll();









  }
}
}
```

```
class Producer implements Runnable {
  public void run() {
    try {
      int i = 0;
      while (!Thread.interrupted()) {
        i = i+1;
        System.out.println("Produced: " + i);
        MessageContainer.put(i);
      }
    } catch (InterruptedException e) {
        System.out.println("Producer has been stopped!");
    }
  }
}

class Consumer implements Runnable {
  public void run() {
    try {
      while (!Thread.interrupted()) {
        int i = MessageContainer.get();
        System.out.println("Consumed: " + i);
      }
    } catch (InterruptedException e) {
        System.out.println("Consumer has been stopped!");
    }
  }
}
```

Erweitern Sie die Methoden `get` und `put` der Klasse `MessageContainer` unter Verwendung der Methoden `wait` und `notifyAll`. Die Methode `get` soll nur dann einen entsprechenden Rückgabewert liefern, wenn eine noch ungelesene Nachricht vorliegt, ansonsten soll sie blockieren, bis eine neue Nachricht vorliegt. Analog dazu soll die Methode `put` eine neue Nachricht nur dann setzen, wenn keine ungelesene Nachricht vorliegt, ansonsten soll sie blockieren.

Aufgabe 4 **Ampelsteuerung**

a)
```
package Ampeln;
public class Ampel extends Thread {
    private boolean fussgangerGruen = false;

    public void run() {
        try {
            while (!isInterrupted()) {
                schalteAmpel();
                sleep(60000);
            }
        } catch (InterruptedException e) {
        }
    }

    public synchronized boolean getFussgaengerAmpelIstGruen() {
        return fussgangerGruen;
    }

    public synchronized boolean getAutoAmpelIstGruen() {
        return !fussgangerGruen;
    }

    private synchronized void schalteAmpel() {
        fussgangerGruen = !fussgangerGruen;
        System.out.println(fussgangerGruen ? "Auto: rot - Fussgänger: grün" : "Auto
        notifyAll();
    }

    public static void main(String[] args) {
        Ampel ampel = new Ampel();
        ampel.start();
        Auto a1 = new Auto("VW", ampel);
        a1.start();
        Fussgaenger f1 = new Fussgaenger("Martin", ampel);
        try {
            sleep(3000);
        } catch (InterruptedException e) {
        }
        f1.start();
        Fussgaenger f2 = new Fussgaenger("Thomas", ampel);
        try {
            sleep(3000);
        } catch (InterruptedException e) {
        }
        f2.start();
        Auto a2 = new Auto("Audi", ampel);
        a2.start();
        Fussgaenger f3 = new Fussgaenger("Frank", ampel);
```

```
        try {
            sleep(3000);
        } catch (InterruptedException e) {
        }
        f3.start();
        Auto a3 = new Auto("Opel", ampel);
        try {
            sleep(3000);
        } catch (InterruptedException e) {
        }
        a3.start();
        a1.join();a2.join();a3.join();f1.join();f2.join();f3.join();
        ampel.interrupt();
    }
}
```

b) Im folgenden ist eine Lösung angegeben, die in sinnvoller Weise Instanzen der beteiligten Objekte generiert und die Threads startet. Für eine korrekte Lösung der Klausur genügte die Instanziierung einer Ampel und zweier Fussgänger, sowie deren Start.

```
package Ampeln;
public class Fussgaenger extends Thread {
    String name;
    Ampel ampel;

    public Fussgaenger(String name, Ampel ampel) {
        this.name = name;
        this.ampel = ampel;
    }

    public void run() {
        System.out.println(name + " wartet");
        synchronized(ampel) {
           while (!ampel.getFussgaengerAmpelIstGruen()){
               try {
                   ampel.wait();
               } catch (InterruptedException e) {
               }
           }
           System.out.println(name + " geht");
}
    }
}
```