

Prof. Dr. Margarita Esponda

# Nichtsequentielle Programmierung, SoeSe 2017

## Übungsblatt 7

TutorIn: Lilli Walter  
Tutorium 6

Boyan Hristov, Sergelen Gongor

4. Juli 2017

---

Link zum Git Repository: <https://github.com/BoyanH/FU-Berlin-ALP4/tree/master/Solutions/Homework7>

### 1. Aufgabe

- a) Um diese Aufgabe richtig zu beantworten, muss man Annahme über welche Threads zu erst bei dem Scheduling-Algorithmus ankommt, obwohl die alle gleich gestartet werden. Das liegt daran, dass bei nicht präemptiven Algorithmen ein Thread nie unterbrochen wird, bevor dieser zu Ende ausgeführt wurde.

1. nicht-präemptive FCFS

Hier machen wir die Annahme, dass der Thread mit 100 Minuten Ausführungszeit erster ankommt. So unterscheiden sich auch die ersten 2 Teilaufgaben

$$\text{Gesamte Ausführungszeit} = 100 + 100 \times 1 = 200$$

$$\text{Durchschnittliche Ausführungszeit} = \frac{200}{101} \approx 1,98 \text{ Minuten}$$

$$\text{Gesamte Wartezeit} = 0 + 99 \times 100 + \sum_{i=0}^{99} i = 990 + \frac{99(99+1)}{2} = 990 + \frac{990}{2} = 1485$$

$$\text{Durchschnittliche Wartezeit} = \frac{1485}{101} \approx 14,7 \text{ Minuten}$$

Damit durchschnittliche Verarbeitungszeit  $\approx 16,68$  Minuten

2. nicht-präemptive SJF

$$\text{Gesamte Ausführungszeit} = 100 \times 1 + 100 = 200$$

$$\text{Durchschnittliche Ausführungszeit} = \frac{200}{101} \approx 1,98 \text{ Minuten}$$

$$\text{Gesammte Wartezeit} = (\sum_{i=0}^{99} i) + 100 = \frac{990}{2} + 100 = 595 \text{ Minuten}$$

$$\text{Durchschnittliche Wartezeit} = \frac{595}{101} \approx 5,89 \text{ Minuten}$$

$$\text{Damit durchschnittliche Verarbeitungszeit} = \frac{199,5+200}{101} \approx 9,82 \text{ Minuten}$$

Unter der Annahme, dass 1. Thread mit 100 Minuten Ausführungszeit als 1. gescheduled wird bekommt man das Ergebniss von FCFS. Wir gehen aber davon aus, dass der Scheduling-Algorithmus schon alle Threads kennt.

3. präemptive Shortest Remaining Time First. Quantum = 1 Minute

Da hier alle Threads gleichzeitig gestartet werden, gehen wir auch davon aus, dass diese in der selben Reihenfolge zu dem Scheduler ankommen. Da alle gleich gestartet werden un keine Threads mit kürzerer Ausführungszeit inzwischen kommen, ist die Ausführung analog zu diese von nicht-präemptive SJF.

Man kann hier die Annahme machen, dass 1. Thread mit Ausführungszeit von 100 Minuten als erster gescheduled wird und dann unterbrochen wird, dann kommt man bei dem Ergebniss von RR. Wie vorher, ist unsere Annahme, dass der Scheduler schon alle Threads kennt.

$$\text{Durchschnittliche Wartezeit} = \frac{5051}{101} \approx 50 \text{ Minuten}$$

$$\text{Durchschnittliche Verarbeitungszeit} \approx 3,96 \text{ Minuten}$$

4. RR-Scheduling. Quantum = 1 Minute

Bei Quantum gleich eine Minute werden alle Threads mit kleine Anforderungen gleich auf der ersten Runde ausgeführt und am Ende bleibt nur das eine Thread noch laufen. Um die Berechnung interessanter zu machen gehen wir davon aus, dass der Thread mit 100 Minuten Ausführungszeit als erster gestartet wurde.

$$\begin{aligned} \text{Gesammte Ausführungszeit} &= \text{Ausführungszeit für 1. Thread} + \text{Ausführungszeit für restliche 100 Threads} + \text{Restausführungszeit für 1. Thread} = \\ &= 1 + (100 \times 1) + 99 = 200 \end{aligned}$$

$$\text{Durchschnittliche Ausführungszeit} = \frac{200}{101} \approx 1,98 \text{ Minuten}$$

$$\begin{aligned} \text{Gesammte Wartezeit} &= \text{Initiale Wartezeit für 1. Thread (0)} + \text{Wartezeiten für einzelne Threads mit Ausführungszeit 1} + \text{Wartezeit für 1. Thread} = \\ &= 0 + (\sum_{i=1}^{100} i) + 100 = \frac{10100}{2} + 100 = 5051 \text{ Minuten} \end{aligned}$$

$$\text{Durchschnittliche Wartezeit} = \frac{5051}{101} \approx 50 \text{ Minuten}$$

$$\text{Durchschnittliche Verarbeitungszeit} = \frac{200,5+200}{101} \approx 3,965 \text{ Minuten}$$

- b) Nicht-präemptive FCFS ist ein sehr simpler Algorithmus mit geringsten Overhead. Der ist gut für Scheduling von unabhängige, gleich große Aufgaben, die erst dann fertig sind, wenn alle fertig sind. Z.B. bei Verarbeitung und Analyse von Daten an einem Server.

Nicht-präemptive SJF ist besser für Scheduling von unterschiedlich-aufwändige Aufgaben, wo wir relativ sicher sind, dass keine neuen Aufgaben inzwischen kommen. Der Algorithmus ist wieder aufwändigbar für Bearbeitung von schon am Anfang bekannte Jobs. Dieser Algorithmus funktioniert aber wesentlich besser für Jobs mit unterschiedliche Ausführungszeit.

Präemptive Shortest Remaining Time First ist schon deutlich besser für simple Betriebssysteme. Hier können Jobs mit unterschiedlichen Anforderungen gut behandelt werden, neu gekommene Threads mit kurze Ausführungszeiten müssen nicht lange warten.

Mit Round-Robin kann man aber deutlich besser Scheduler für Threads mit unbekannte Ausführungszeiten bauen. Das liegt daran, dass jeden Thread einen festen Zeitfenster kriegt, in dem dieser fertig sein soll, sonst wartet er auf die nächste Runde. Der Algorithmus eignet sich gut für Systemen, wo Ausführungszeit von einzelne Threads schwer zu schätzen ist.

## 2. Aufgabe

- a) FCFS - Es könnte sein, dass einige Threads lange auf den aufwändigen Threads vor denen warten, die werden aber alle irgendwann ausgeführt, da der Algorithmus mit einer Schlange funktioniert.
- b) SJF - Hier können aufwändigere Threads sehr leicht verhungern, in dem immer wieder neue Threads kommen mit sehr geringe Ausführungszeit.
- c) RR - hier könnte es wieder sein, dass einige Threads mit längeren Ausführungszeiten mehrmals gestartet / weiter ausgeführt werden bevor diese fertig werden, Threads werden aber nicht verhungern.
- d) O(1) früheres Linux-Scheduling - Threads mit geringere Priorität haben nur halb so große Quanta, es könnte wieder sein dass einige deutlich länger verarbeitet werden, Threads verhungern aber wieder nicht.
- e) Hybrid Lottery Scheduling - da alle Prozesse eine Mindestanzahl von Losen haben, verhungern keine Prozesse / Threads.
- f) Completely Fair Scheduling - die Frage kann mit Argumenten aus der Vorlesung leider nicht beantwortet werden. Der Fair Scheduler benutzt aber die Zeit, in dem ein Prozess zu der Scheduler gekommen ist, als auch die letzte Ausführungszeit um die Prioritäten von älteren Prozessen zu erhöhen, vermeidet so erfolgreich das Verhungern von Prozessen.

Quelle → <https://stackoverflow.com/questions/39725102>

## 3. Aufgabe

Die Bilder kann man in GitHub besser anschauen

- a) RMS

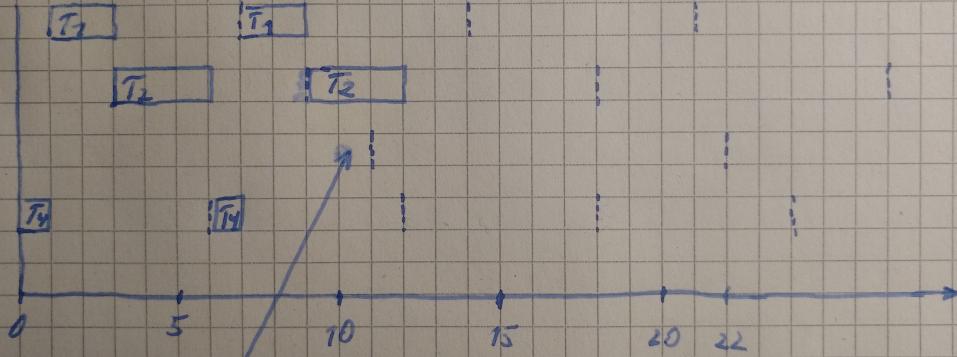
RMS - Prioritäten umgekehrt proportional zur Periode  $p$

$$T_1 = (0, 2, 7, 2)$$

$$T_2 = (0, 3, 8, 8)$$

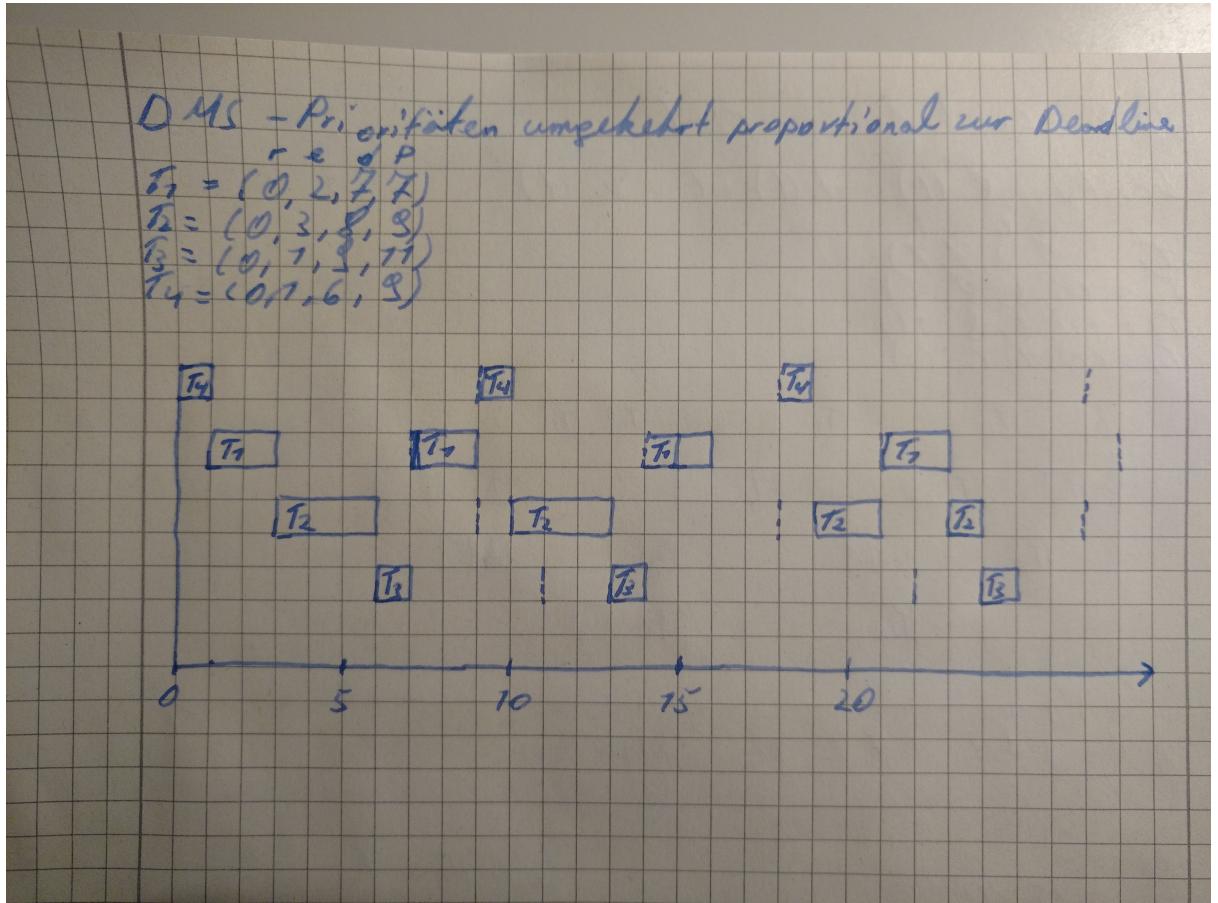
$$T_3 = (0, 7, 11, 11)$$

$$T_4 = (0, 7, 6, 6)$$

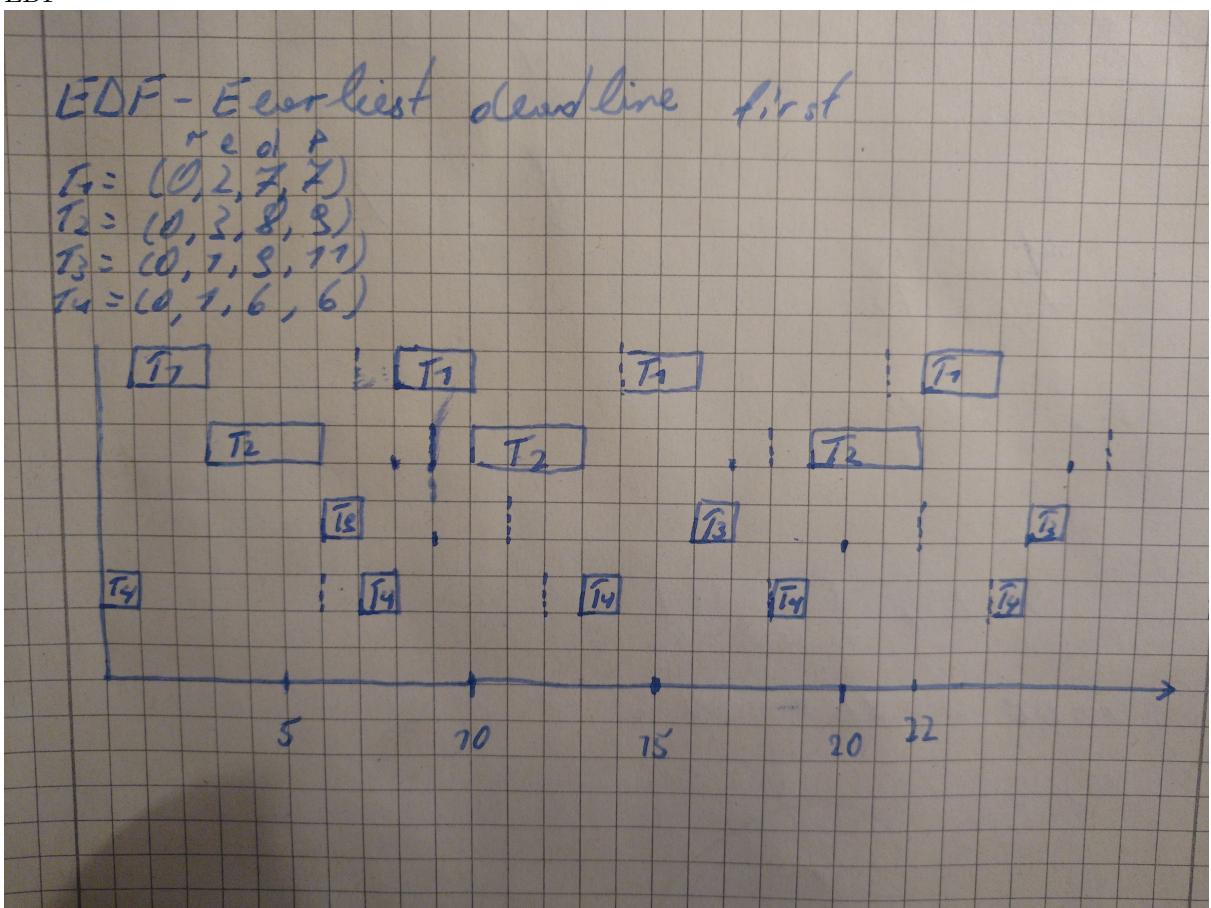


↳  $T_3$  hat seine Deadline verpasst!

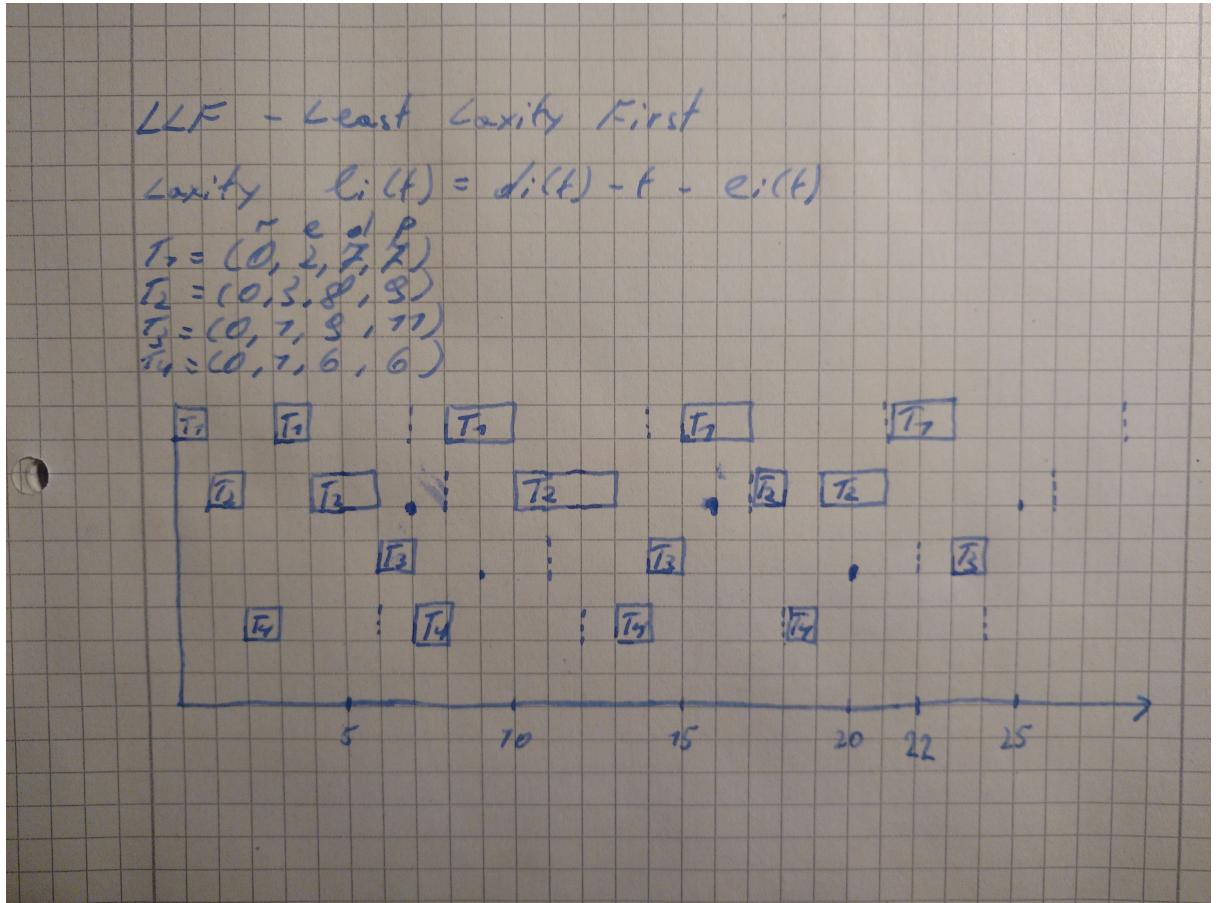
b) DMS



c) EDF



d) LLF



#### 4. Aufgabe

Ja, das System ist realisierbar. In einer Sekunde gibt es 1000 Millisekunden, deswegen müssen wir ein Bild je  $\frac{1000}{25} = 40$  Millisekunden fertig kriegen. Wir können mit dem LLF Algorithmus das Problem leicht lösen. Wenn man die Zeit in 5-Millisekunden-Abschnitte zerteilt, wird in der ersten Millisekunde die Sprachverbindung bearbeitet und in den restlichen 4 die Bildbearbeitung. So schafft man die ganze Bildbearbeitung in 5 solche Abschnitte (20ms Bearbeitung) und verpasst dadurch keine Sprachverbindung. Das ganze ist aber nur unter der Annahme möglich, dass LLF kleines Overhead hat, was leider nicht der Fall ist. Da es aber um ein weiches Echtzeitsystem geht, können wir es immer realisieren, in dem die Qualität vermindert wird.

## 5. Aufgabe

Die Aufgabe wurde unter der Annahme gelöst, dass das System eine weiche Echtzeitsystem ist. Sonst könnte man leicht eine Fehler in dem Task Klasse werfen und diese in dem Scheduler abfangen und terminieren.

```
1 package fu.alp4;
2
3 public class Main {
4
5     public static void main(String[] args) {
6
7         Task[] tasksForSimulation = {
8             new Task(0,2,7,7),
9             new Task(0,3,9,9),
10            new Task(0,1,11,11),
11            new Task(0,1,6,6)
12        };
13        RealTimeScheduler scheduler = new RealTimeScheduler(tasksForSimulation, 22);
14        scheduler.simulateRMS();
15    }
16}
```

```
1 package fu.alp4;
2
3 import java.util.Arrays;
4
5 public class RealTimeScheduler {
6
7     private Task[] managedTasks;
8     private int simulationTime;
9     private Task latestExecuted;
10
11    public RealTimeScheduler(Task[] tasks, int st) {
12        this.managedTasks = tasks;
13        this.simulationTime = st;
14    }
15
16    public void simulateRMS() {
17        this.setPrioritiesRMS();
18
19        System.out.println("\nSimulation starting!\n\n");
20
21        while(this.simulationTime > 0) {
22
23            boolean taskAlreadyExecutedInCurrentLoop = false;
24
25            // managed tasks already sorted by priority
26            for(Task task : this.managedTasks) {
27                if (task.getReady() && !taskAlreadyExecutedInCurrentLoop) {
28
29                    if (this.latestExecuted == null) {
30                        this.latestExecuted = task;
31                    } else if (this.latestExecuted != task) {
32                        System.out.printf("Task with #id %s was interrupted in favour of
task with #id %s\n\n",
33                                         this.latestExecuted.getId(), task.getId());
34                        this.latestExecuted = task;
35                    }
36
37                    task.execute();
38                    taskAlreadyExecutedInCurrentLoop = true;
39
40                } else {
41                    task.keepBlocked();
42                }
43            }
44
45        }
46    }
47}
```

```

43         }
44
45         this.simulationTime--;
46     }
47
48 }
49
50 private void setPrioritiesRMS() {
51     // Sort managedTask by period
52     Arrays.sort(this.managedTasks, (a, b) -> a.getPeriod() < b.getPeriod() ? -1 :
53             a.getPeriod() == b.getPeriod() ? 0 : 1);
54
55     // and assign priorities equal to reverse sorting number (e.g. the task with lowest
56     // period will be first and
57     // will receive highest priority equal to the number of managed tasks)
58     for (int i = 0; i < this.managedTasks.length; i++) {
59         Task currentTask = this.managedTasks[i];
60         currentTask.setPriority(this.managedTasks.length - i);
61         System.out.printf("Task id: %s; Period: %s; Priority: %s\n",
62                           currentTask.getId(), currentTask.getPeriod(), currentTask.getPriority());
63     }
64 }
65 }
```

```

1 package fu.alp4;
2
3 public class Task {
4
5     static int idCounter;
6
7     private int release;
8     private int execution;
9     private int executionLeft;
10    private int timePassed;
11    private int deadline;
12    private int deadlineAfter;
13    private int period;
14
15    private int priority;
16    private int id;
17
18    /**
19     * A Task manages itself in terms of setting its own deadlineAfter and executionLeft.
20     * The idea is to let the scheduler-class only implement the scheduler logic, as if it
21     * was
22     * gathering the data from the Process-Control-Block
23     */
24
25    public Task(int r, int e, int d, int p) {
26        this.release = r;
27        this.execution = e;
28        this.executionLeft = release == 0 ? e : 0;
29        this.deadline = d;
30        this.deadlineAfter = d;
31        this.period = p;
32        this.timePassed = 0;
33        this.setPriority(idCounter); // initially set priority = highest possible, will be
34        // changed later on
35
36        /**
37         * Priority can be between 1/n and n/n for n tasks in the scheduler.
38         */
39
40        this.id = ++idCounter;
41    }
42 }
```

```

42     public int getPeriod() {
43         return period;
44     }
45
46     public boolean getReady() {
47         return this.executionLeft > 0;
48     }
49
50     public int getPriority() {
51         return priority;
52     }
53
54     public int getId() {
55         return this.id;
56     }
57
58     public void setPriority(int priority) {
59         this.priority = priority;
60     }
61
62     public void execute() {
63         this.executionLeft--;
64         this.passTime();
65         System.out.printf("Task with id %s and priority %s executed. \n\t Execution left: %s; Deadline after: %s ;Time passed: %s\n\n",
66             this.id, this.getPriority(), this.executionLeft, this.deadlineAfter, this.timePassed);
67     }
68
69     public void keepBlocked() {
70         this.passTime();
71     }
72
73     private void passTime() {
74         this.timePassed++;
75         this.deadlineAfter--;
76
77         if (this.executionLeft > 0 && this.deadlineAfter == 0) {
78             System.out.printf("Task with id %s and priority %s missed it's deadline!\n\n",
79                 this.id, this.getPriority());
80         } else if (this.timePassed % this.period == 0 || this.timePassed == this.release) {
81             this.deadlineAfter = this.deadline;
82             this.executionLeft = this.execution;
83         }
84     }

```

## 6. Aufgabe

- e) Yeah, I read the documentation, I promise! :)
- f) Auf Zeile 40 kriegen wir ein byte[] von dem MD5 Hash als einen Byte Array. Das wollen wir aber gerne als String[] haben, damit wir es leichter speichern können (als Klartext) oder mit anderen Hasches vergleichen können. Da aber der byte[] ein Array von signed Bytes ist (also negative auch) und Java hardkodierte hexadecimale Nummer immer als positiv annimmt können wir so festdefinieren, dass wir da positive 16bit Nummer haben (mit array[i] & 0xFF). Da aber in MD5 definiert ist, dass jedes byte 2 Positionen nimmt (2 Digits in Hexadezimalsystem) verordern wir das Ergebniss mit 0x100, oder (0001 0000 0000)<sub>2</sub> damit wir für 0x01 entsprechend "01" bekommen und nicht "1". Alle 2-Digit Hexadezimalzahlen, die von je ein Byte konvertiert werden, werden zu dem Endergebniss appended, was am Ende zurückgegeben wird.

```

g)
1 package hashCracker;

4 import fx.ResultPool;

6 public class SimpleCracker implements HashCracker {

8     ResultPool resPool;
9     boolean stopped;

11    @Override
12    public void start(String[] wordList, String[] hashList, ResultPool resPool) {
14
15        this.stopped = false;
16
17        for (int i = 0; i < wordList.length; i++) {
18            for (int j = i; j < wordList.length; j++) {
19                String wordA = wordList[i];
20                String wordB = wordList[j];
21                String variantA = wordA + wordB;
22                String variantB = wordB + wordA;
23
24                for (int k = 0; k < hashList.length; k++) {
25                    if (MD5(variantA).equals(hashList[k])) {
26                        resPool.pushPassword(variantA);
27                    } else if (MD5(variantB).equals(hashList[k])) {
28                        resPool.pushPassword(variantB);
29                    }
30
31                    if (this.stopped) {
32                        return;
33                    }
34                }
35            }
36        }
37
38    @Override
39    public void stop() {
40        this.stopped = true;
41    }
42
43    public static String MD5(String md5) {
44        try {
45            java.security.MessageDigest md = java.security.MessageDigest.getInstance("MD5");
46            byte[] array = md.digest(md5.getBytes());
47            StringBuffer sb = new StringBuffer();
48            for (int i = 0; i < array.length; ++i) {
49                sb.append(Integer.toHexString((array[i] & 0xFF) | 0x100).substring(1,
50            3));
51            }
52            return sb.toString();
53        } catch (java.security.NoSuchAlgorithmException e) {
54        }
55        return null;
56    }
57
58 }

```

- h) Wir haben die einfache Lösung genommen und versucht, diese so gut wie möglich auf 4 Prozessoren skalieren zu lassen, da wir davon ausgegangen sind, dass die meiste Laptops heutzutage genau so viele CPU Cores haben bzw. so viele simulieren durch Hyperthreading.

Dabei haben wir gesehen, dass wenn wir die Äußere Schleife durch 4 teilen und das separat rechnen

lassen, die Threads die ein größeres Anfangsindex haben weniger Werte ausrechnen (da die innere Schleife von  $i$  anfängt) und nämlich genau  $\frac{n}{4}$  bei  $n$  mögliche Wörter. Da wir aber keine perfekte Aufteilung finden könnten, haben wir gemeint, dass es relativ ok ist, dass der 1 Thread  $\frac{n}{8}$  weniger Elemente in der äußeren Schleife ausrechnen soll und der letzten entsprechend so viel mehrere.

Code Nice try, klappte aber eigentlich nicht so gut.

```

1 package hashCracker;
2
3 import fx.ResultPool;
4
5 public class CrackerThread extends Thread {
6
7     String[] wordList;
8     String[] hashList;
9     ResultPool resPool;
10    int startIndex;
11    int endIndex;
12    boolean stopped;
13
14    public CrackerThread(String[] wordList, String[] hashList, ResultPool resPool, int
15        i) {
16        this.wordList = wordList;
17        this.hashList = hashList;
18        this.resPool = resPool;
19        this.startIndex = (int) Math.floor(i*wordList.length/4);
20        this.endIndex = (int) Math.ceil(startIndex + wordList.length/4);
21        this.stopped = false;
22
23        // spaghetti part here, just wanted to try this prototype solution
24        // could be fixed at some point
25
26        if (i == 3) {
27            this.startIndex -= (int) Math.ceil(wordList.length / 8);
28        } else if (i != 0) {
29            this.startIndex -= (int) Math.ceil(wordList.length / 8);
30            this.endIndex -= (int) Math.ceil(wordList.length / 8);
31        } else {
32            this.endIndex -= (int) Math.ceil(wordList.length / 8);
33        }
34
35        if (this.startIndex < 0) {
36            this.startIndex = 0;
37        }
38
39        if (this.endIndex > this.wordList.length) {
40            this.endIndex = this.wordList.length;
41        }
42    }
43
44    public void stopCalc() {
45        this.stopped = true;
46    }
47
48    @Override
49    public void run() {
50
51        int count = 0;
52        int gausNumber = this.endIndex - 1 - startIndex;
53        String[] pairs = new String[gausNumber*(gausNumber+1)];
54
55        for (int i = this.startIndex; i < this.endIndex; i++) {
56            for (int j = i; j < wordList.length; j++) {
57
58                String[] variants = {
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
627
628
629
629
630
631
632
633
634
635
635
636
637
637
638
638
639
639
640
640
641
641
642
642
643
643
644
644
645
645
646
646
647
647
648
648
649
649
650
650
651
651
652
652
653
653
654
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
151
```

```
58             wordList[i] + wordList[j],
59             wordList[j] + wordList[i],
60             wordList[i],
61             wordList[j]
62         };
63
64         for (String hash : hashList) {
65             for (String variant : variants) {
66                 if (MD5(variant).equals(hash)) {
67                     resPool.pushPassword(variant);
68                 }
69             }
70
71             if (this.stopped) {
72                 return;
73             }
74         }
75     }
76 }
77
78 }
79
80 public static String MD5(String md5) {
81     try {
82         java.security.MessageDigest md = java.security.MessageDigest.getInstance("MD5");
83         byte[] array = md.digest(md5.getBytes());
84         StringBuffer sb = new StringBuffer();
85         for (int i = 0; i < array.length; ++i) {
86             sb.append(Integer.toHexString((array[i] & 0xFF) | 0x100).substring(1,
87             3));
88         }
89         return sb.toString();
90     } catch (java.security.NoSuchAlgorithmException e) {
91     }
92     return null;
93 }
```

```
1 package hashCracker;  
2  
3  
4 import fx.ResultPool;  
5  
6 public class BetterCracker implements HashCracker {  
7  
8     ResultPool resPool;  
9     CrackerThread[] crackerThreads;  
10  
11    @Override  
12    public void start(String[] wordList, String[] hashList, ResultPool resPool) {  
13  
14        this.crackerThreads = new CrackerThread[4];  
15  
16        for (int i = 0; i < 4; i++) {  
17            this.crackerThreads[i] = new CrackerThread(wordList, hashList, resPool, i)  
18        ;  
19            this.crackerThreads[i].start();  
20        }  
21  
22        for (int t = 0; t < this.crackerThreads.length; t++) {  
23            try {  
24                this.crackerThreads[t].join();  
25                this.stop();  
26            } catch (InterruptedException e) {  
27                e.printStackTrace();  
28            }  
29        }  
30    }  
31}
```

```
29     }
30
31     @Override
32     public void stop() {
33
34         for (CrackerThread thread : this.crackerThreads) {
35             thread.stopCalc();
36         }
37     }
38 }
```