

Prof. Dr. Margarita Esponda

# Nichtsequentielle Programmierung, SoeSe 2017

## Übungsblatt 7

TutorIn: Lilli Walter  
Tutorium 6

Boyan Hristov, Sergelen Gongor

27. Juni 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

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

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

$$\text{Gesamte Wartezeit} = 0 + 99 \times 100 + \sum_{i=0}^{99} i = 990 + \frac{99(99+1)}{2} = \frac{1999}{2}$$

$$\text{Durchschnittliche Wartezeit} = \frac{1999}{2 \times 101} = \frac{1999}{202} \approx 9,896 \text{ Minuten}$$

$$\text{Damit durchschnittliche Verarbeitungszeit} = \frac{200 + \frac{1999}{2}}{101} \approx 11,876 \text{ Minuten}$$

2. nicht-präemptive SJF

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

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

$$\text{Gesammte Wartezeit} = (\sum_{i=0}^{99} i) + 100 = \frac{199}{2} + 100 = 199,5 \text{ Minuten}$$

$$\text{Durchschnittliche Wartezeit} = \frac{199,5}{101} \approx 1,975 \text{ Minuten}$$

$$\text{Damit durchschnittliche Verarbeitungszeit} = \frac{199,5+200}{101} \approx 3,955 \text{ Minuten}$$

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.

$$\text{Durchschnittliche Wartezeit} = \frac{199,5}{101} \approx 1,975 \text{ Minuten}$$

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

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.

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

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

$$\begin{aligned} \text{Gesammte Wartezeit} &= \text{Initiale Wartezeit für 1. Thread (0)} + \text{Wartezeiten für einzelne Threads mit Ausführungszeit 1} + \text{Wartezeit für 1. Thread} = \\ &= 0 + (\sum_{i=1}^{100} i) + 100 = \frac{201}{2} + 100 = 199,5 \text{ Minuten} \end{aligned}$$

$$\text{Durchschnittliche Wartezeit} = \frac{200,5}{101} \approx 1,985 \text{ Minuten}$$

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

- 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

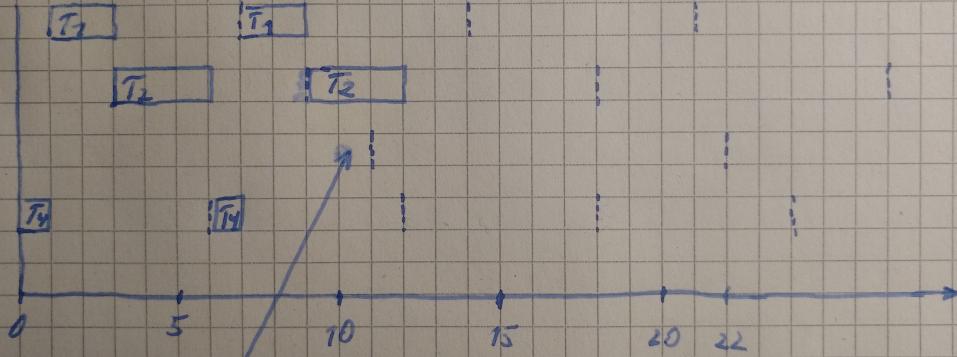
RMS - Prioritäten umgekehrt proportional zur Periode  $p$

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

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

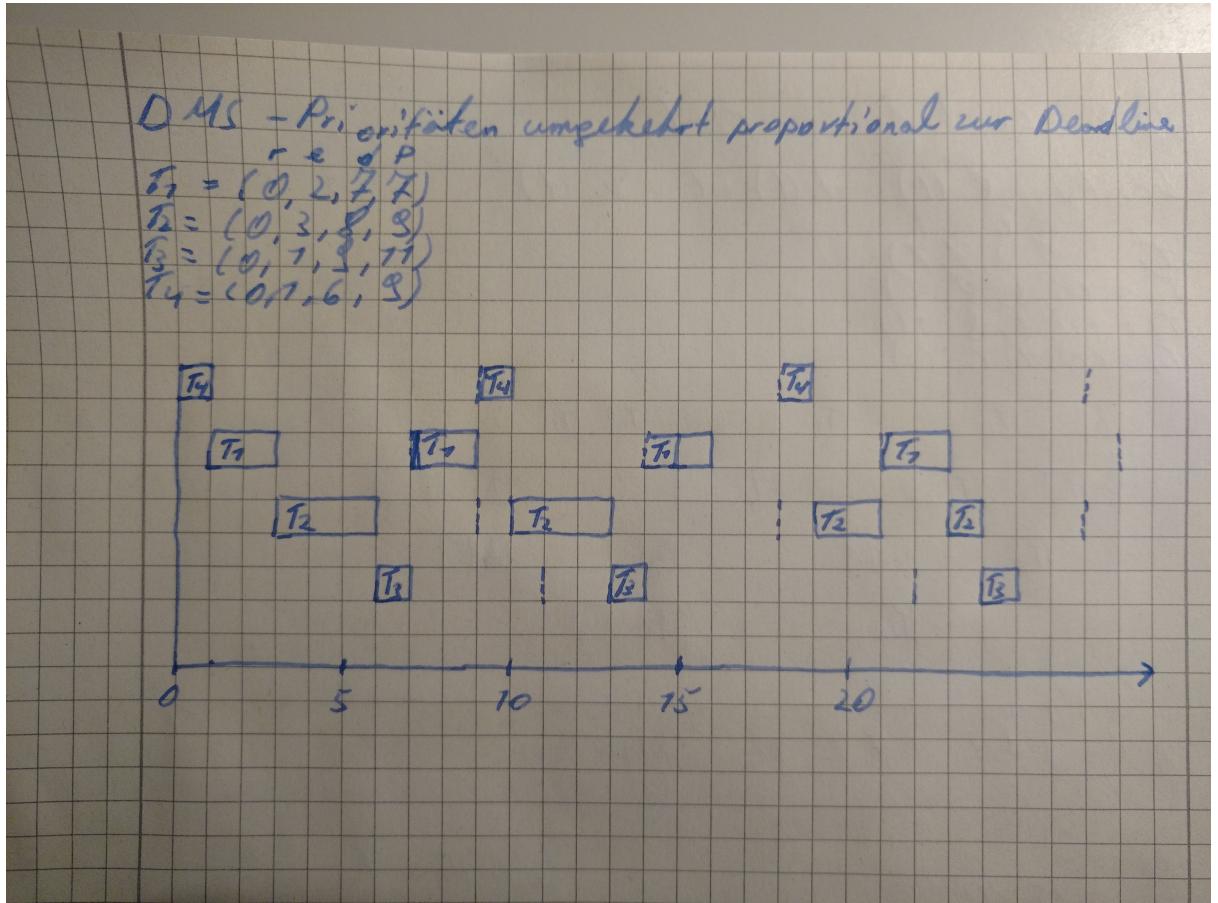
$$T_3 = (0, 7, 11, 11)$$

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

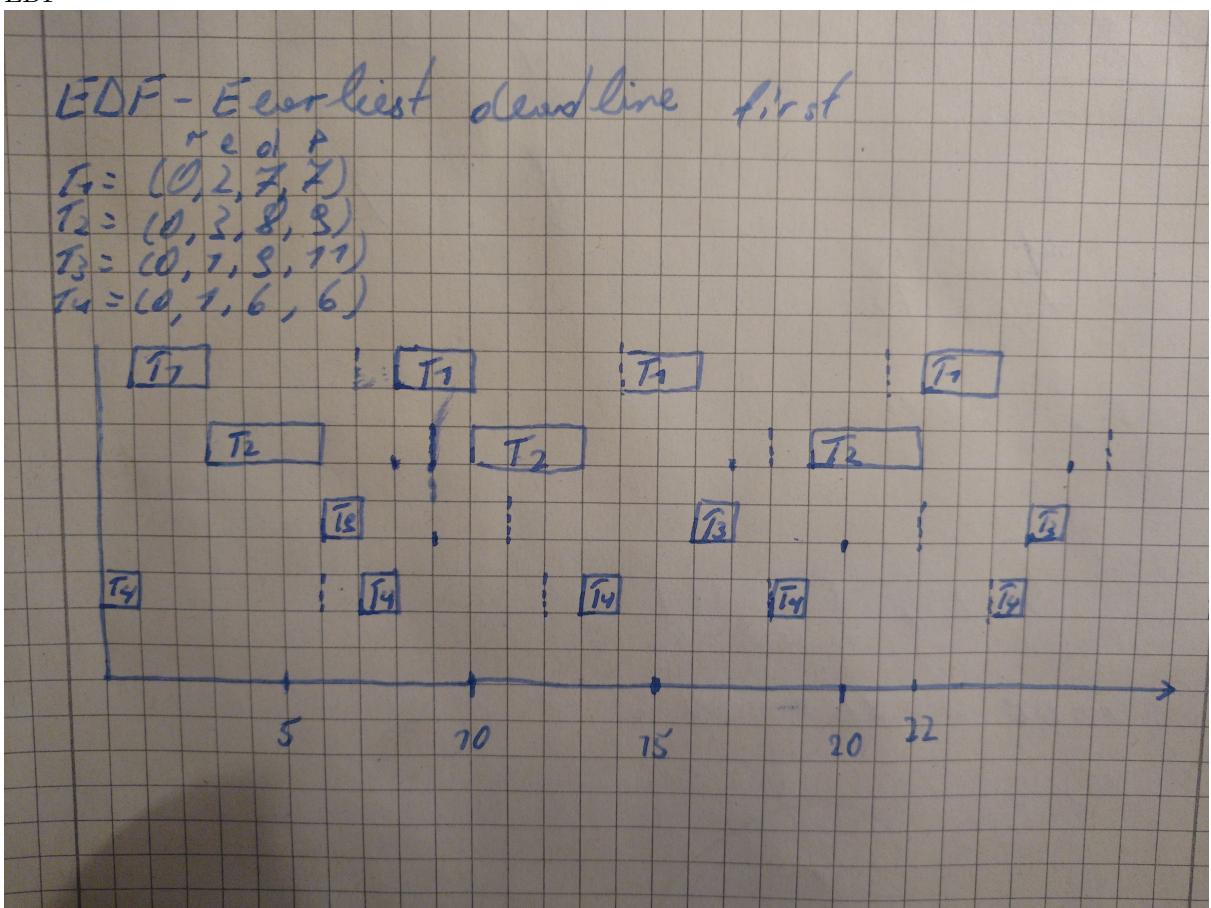


↳  $T_3$  hat seine Deadline verpasst!

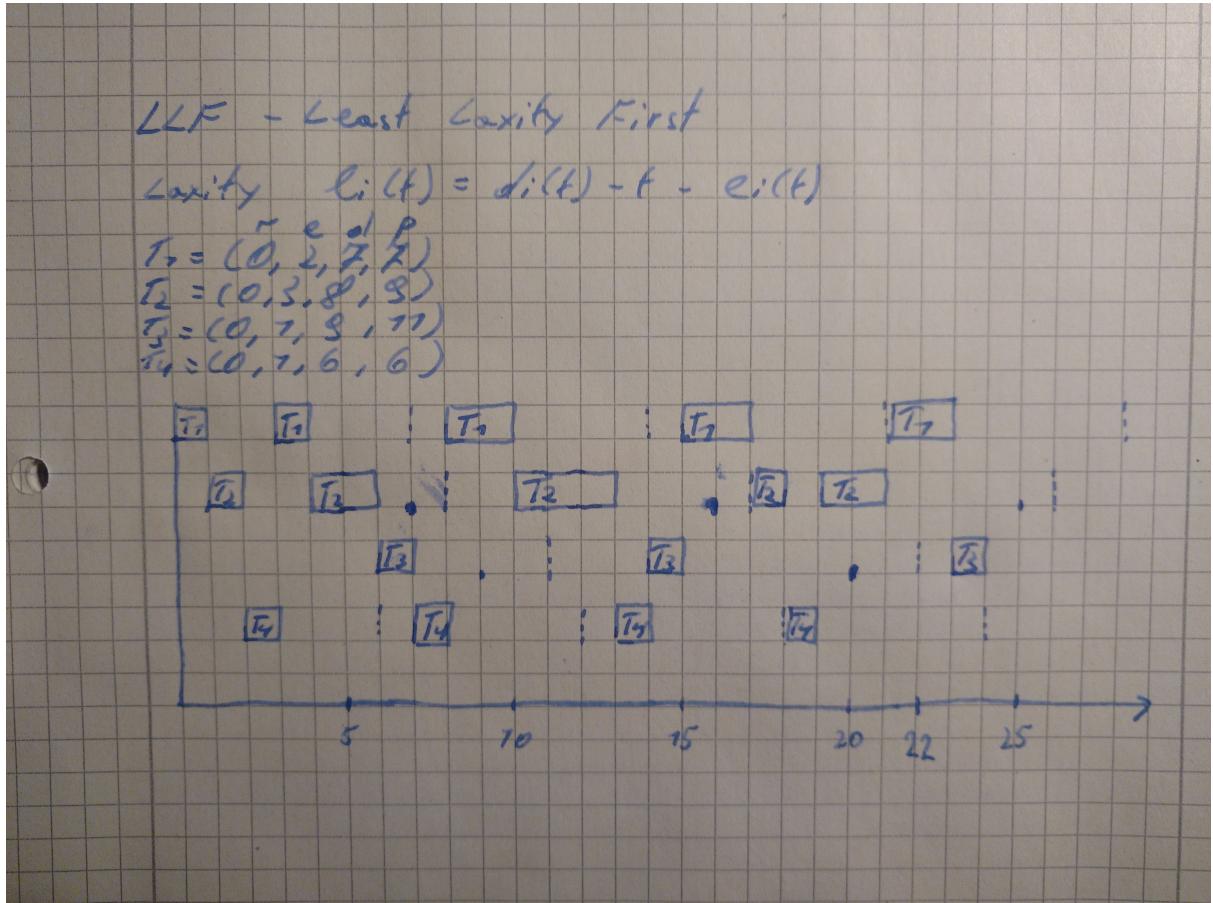
b) DMS



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;
3
3 public class Main {
5
5     public static void main(String[] args) {
7
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;
3
3 import java.util.Arrays;
5
5 public class RealTimeScheduler {
7
7     private Task[] managedTasks;
8     private int simulationTime;
9     private Task latestExecuted;
11
11     public RealTimeScheduler(Task[] tasks, int st) {
12         this.managedTasks = tasks;
13         this.simulationTime = st;
14     }
16
16     public void simulateRMS() {
17         this.setPrioritiesRMS();
19
19         System.out.println("\nSimulation starting!\n\n");
21
21         while(this.simulationTime > 0) {
23
23             boolean taskAlreadyExecutedInCurrentLoop = false;
25
25             // managed tasks already sorted by priority
26             for(Task task : this.managedTasks) {
27                 if (task.getReady() && !taskAlreadyExecutedInCurrentLoop) {
29
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                 }
37
37                 task.execute();
38                 taskAlreadyExecutedInCurrentLoop = true;
39
40             } else {
41                 task.keepBlocked();
42             }
43
44         }
45
45     }
46
46 }
```

```

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     }
85 }
```