

## Embedded Systems

### 3. Extended Exercise

## Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>                              | <b>2</b>  |
| <b>2</b> | <b>Preparation</b>                               | <b>2</b>  |
| 2.1      | Basics of Dynamic Scheduling . . . . .           | 2         |
| 2.2      | Dynamic Scheduling . . . . .                     | 3         |
| 2.2.1    | Algorithms for Tasks without Deadlines . . . . . | 4         |
| 2.2.2    | Algorithms for Tasks with Deadlines . . . . .    | 4         |
| 2.2.3    | Algorithms for Periodic Tasks . . . . .          | 4         |
| <b>3</b> | <b>Experiment</b>                                | <b>5</b>  |
| 3.1      | Preparation . . . . .                            | 5         |
| 3.1.1    | How it works . . . . .                           | 8         |
| 3.2      | Scheduling of Tasks without Deadlines . . . . .  | 9         |
| 3.3      | Scheduling of Tasks with Deadlines . . . . .     | 10        |
| 3.4      | Scheduling for Periodic Tasks . . . . .          | 11        |
| <b>A</b> | <b>References</b>                                | <b>12</b> |
| <b>B</b> | <b>Sourcefile <code>scheduleFCFS.cc</code></b>   | <b>13</b> |

# 1 Introduction

Digital embedded systems are ubiquitous in our daily lives and perform a variety of tasks. They often consist of a combination of hardware and software components to perform specific tasks (operations, processes, tasks, etc.). Depending on the application, an embedded system or different parts of it can be classified, for example in *state-oriented*, *event-oriented* or *data-flow-oriented*. Multimedia applications, for example, are data flow-oriented, since individual images are processed periodically. The type and scope of the calculations are independent of the multimedia data to be processed and can be planned statically. The planning includes the allocation of execution times (*scheduling*) and the *binding* to processing units (resources) and usually takes place at design time. These so-called *offline algorithms* for determining a schedule and a binding also have the advantage that they increase the quality and predictability of achievable solutions in many cases. Known scheduling algorithms [4] for the optimization of latency, costs or other variables range from efficient algorithms (e.g. B. ASAP or ALAP methods) and heuristics (e.g. LIST scheduling) to computation-intensive exact methods (e.g. using integer linear programming).

If the individual tasks of a system are not known in advance, the scheduling can only be determined at runtime, i.e. during system operation. This means that at a certain point only existing tasks are known, while problems that will arise in the future are unknown and thus may create tasks dynamically. An example is the highly dynamic query to a search engine. In real-time systems, there is also the fact that in addition to the dynamic appearance, there are also requirements for the latest completion (*Deadline*) of a task.

Dynamic scheduling, partly with real-time requirements, is the subject of this exercise description.

## 2 Preparation

The tools used in this experiment run under Linux on the Chair's workstations. We therefore assume that you are at least familiar with the basic principles of this operating system.

***Please prepare for the experiment*** by reading the description and preparing at home as best you can. Furthermore, you should deal with the topic of dynamic scheduling (chapter 4.7 in [4]) in advance and have already worked through the task sheets 10+11 of the course Embedded Systems [5]. It is advisable to bring these, along with the solution, to the lab. Furthermore, knowledge of the programming language C++ [2] or C is helpful, but not mandatory.

### 2.1 Basics of Dynamic Scheduling

The handling of dynamically occurring tasks in a computing system is typically performed by an operating system [3] with the following responsibilities:

- quasiparallel execution of tasks on one or multiple processors,
- sharing of resources, *adherence to deadlines, minimizing waiting times and ensuring fairness during execution*,

A simplified state machine of task management for a single processor system is shown in Figure 1. It contains the three states *ready*, *running* and *waiting*. New tasks created in an environment are put into the state *ready* and wait there until they are selected and assigned to the processor for processing. This process is called *dispatch*. Processing takes place in the *running* state until either (a) the task



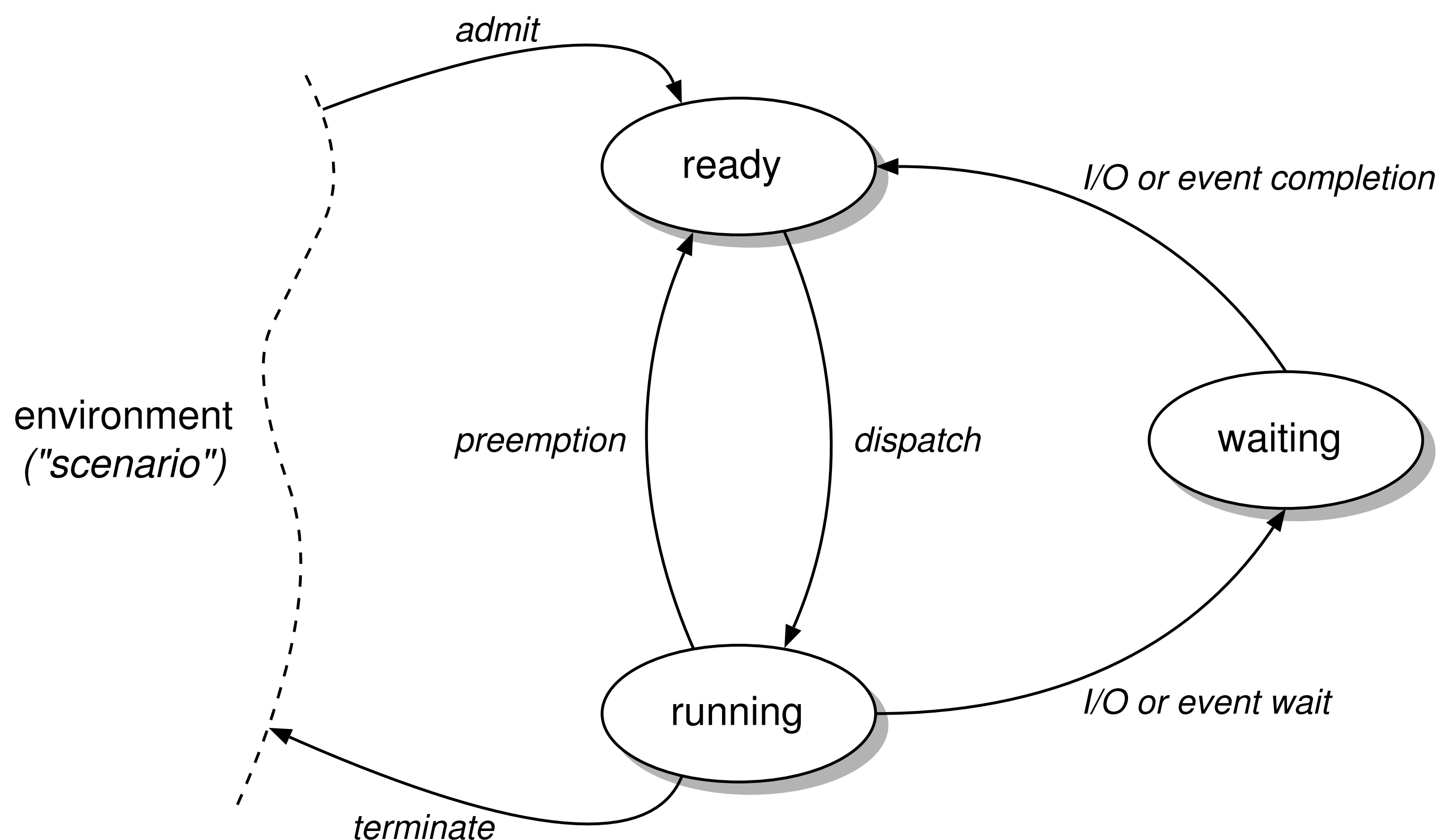


Figure 1: Task management state machine.

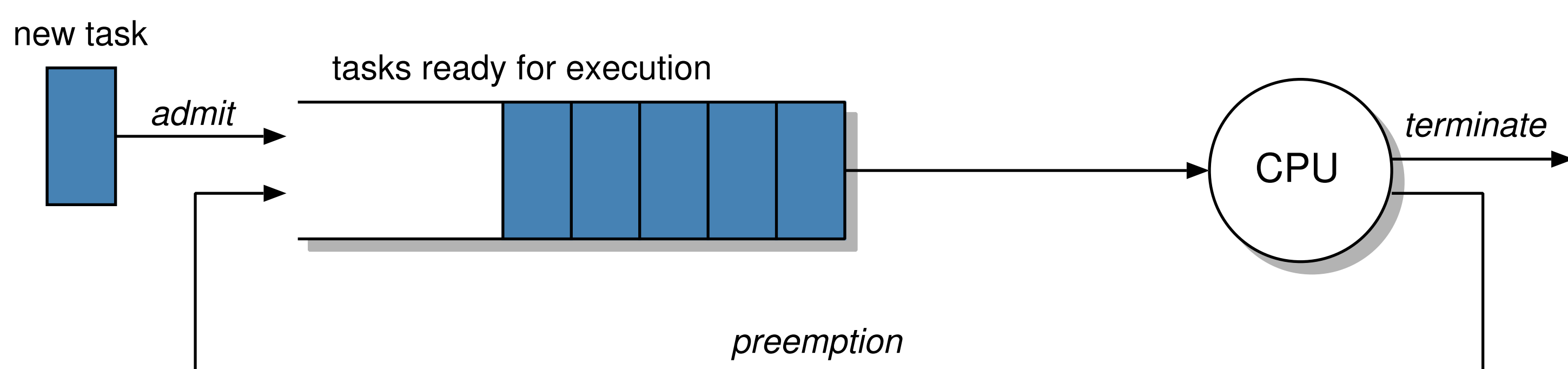


Figure 2: Task scheduling queue.

has been processed and terminated, (b) it is preempted for reasons of priority or fairness, or (c) it is waiting for an input or event. In case of (b) the task is put back to the waiting state *ready*, in case of (c) it goes to the state *waiting* until the input or the event is done, and from there back to the executable tasks (state *ready*).

In the following we will deal mainly with the management of the executable tasks in the state *ready* and their scheduling, the transfer into and the possible preemption from the state *running*. The consideration of inputs or events (state *waiting*) is not part of the exercise.

## 2.2 Dynamic Scheduling

A *queue* can be used to manage the ready to run tasks, see figure 2. Essential criteria of different scheduling algorithms are the order in which the tasks are taken from the queue and whether running tasks can be preempted by the CPU (*Central Processing Unit*) before their completion.

The tasks  $T_i$  to be executed are each characterized by their *release time*  $t_r(T_i)$  and their known execution time  $d_i$ . Optionally, the tasks can have a deadline  $t_d(T_i)$  and possibly dependencies among each other. Another option is the periodic occurrence of a task at a given period (iteration interval)  $P(T_i)$ . The deadlines are also replicated at the interval of the period, but can also have an offset.

According to the above options, we differentiate between the following scheduling algorithms:

- Tasks without deadlines,

- Tasks with deadlines,
- periodic Tasks.

### 2.2.1 Algorithms for Tasks without Deadlines

We will consider these scheduling algorithms for tasks without deadlines:

**FCFS (first come, first served):** According to the motto „First come, first served“, the tasks are processed one after the other. This is fair, because everyone has to queue at the back of the queue. If, however, there are already a lot of tasks with long calculation times in the queue, it can take a long time for a new task to be assigned to the CPU.

**SJF (shortest job first):** Here the tasks are planned according to the motto „The smallest first“. This has the advantage that the CPU is released as fast as possible and thus the average waiting time is minimized.

**SRTN (shortest remaining time next):** This is variant of the SJF scheduling algorithm *with* preemption. At any time, the task whose remaining calculation time is minimal is selected and planned.

**PB (priority-based):** Each task has a fixed priority according to which it is selected and planned. The algorithm is also uses preemption.

**RR (round robin):** The problem of *starvation* of a task is avoided by a fixed time quantum  $Q$ , after which the task will be changed at the latest. The executable tasks are put back into the queue again.

### 2.2.2 Algorithms for Tasks with Deadlines

We will consider these scheduling algorithms for tasks with deadlines:

**EDD (earliest due date):** The algorithm minimizes *lateness* by prioritizing tasks with early deadlines. All tasks can not be preempted in their execution and must have the same release time.

**EDF (earliest deadline first):** This scheduling algorithm is similar to the EDD algorithm in its task selection. However, the tasks can have different release times and can be preempted.

**EDF\*:** With this variant, tasks can have dependencies between each other. The individual release times and deadlines are transformed according to the dependencies and then planned using the EDF procedure.

### 2.2.3 Algorithms for Periodic Tasks

We will consider these scheduling algorithms for periodic tasks:

**EDF (earliest deadline first):** See Section [2.2.2](#).

**DM (deadline monotonic):** In this scheduling algorithm, the deadlines of tasks do not necessarily have to be equal to their periods. Each task is assigned a priority. The smaller the relative deadline  $t_d^*(v_i)$ , the higher the priority. Higher priority tasks preempt lower priority tasks.



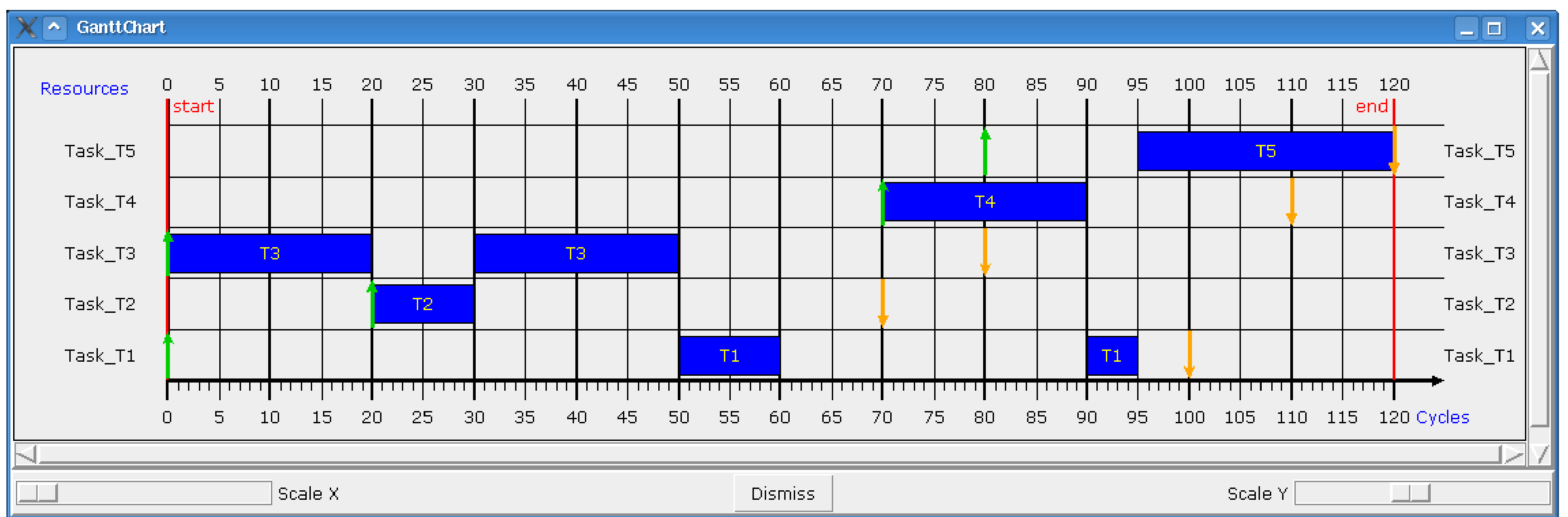




Figure 3: Screenshot of the Gantt chart display program.

## 3 Experiment

The goal of this exercise is to implement the dynamic scheduling procedures described above and to calculate some of their characteristics, such as the flow time. But don't worry, you don't have to start from scratch! There is already an experimental framework available which can be used and expanded and is described in the following section.



### 3.1 Preparation

*Login* to a lab computer. To perform the trial you will need a *Shell* (command line interpreter) and an editor (e.g. *kate* or *vi*) to edit the source files. Commands, that you have to enter into the *Shell*, are marked in the following by a  symbol. Tasks for which you need an editor are identified by a  symbol at the margin. The directory `ueb03` contains all necessary files for the attempt. Change to this working directory.

 `cd ~/ueb03`

An essential part of the entire experiment will be the visualization of flow charts. A program for displaying Gantt charts is available for this purpose. Start the program for an exemplary flow chart with the following command.

 `./showGanttChart schedule_ref.txt`

After starting and making some adjustments to the sliders, you should see a similar graph as in Figure 3. It shows the schedule of five *Tasks*  $T_1, \dots, T_5$  over time (*cycles*). For example, task  $T_3$  is executed in the interval from 0 to 50, but with a preemption for 10 cycles. During this time task  $T_2$  is processed. Green, ascending arrows  visualize the release times and orange, downward arrows  visualize the deadlines of the tasks.

Close the Gantt chart and then get familiar with the other files and sub directories in the working directory.

 `ls -al`

The most important files and subdirectories have the following meaning:

- `gantt.tcl` – A program in the scripting language Tcl/Tk for visualizing Gantt charts.
- `HelpFunctions.cc` – Contains some auxiliary functions, for example for command line output or for generating text files containing the schedule.
- `MainProgram.cc` – Contains the main program.
- `Makefile` – Allows the automatic translation of all required files.
- `scenarios/` – This subdirectory contains some scenarios with predefined tasks (including processing times, arrival times, etc.).
- `Schedule{.h, .cc}` – Contain the class `Schedule`, which contains a schedule for a Task (see below).
- `scheduler/` – This subdirectory contains a source file for each of the different scheduling algorithms. For example, the file `scheduleFCFS.cc`, implements the FCFS procedure.
- `showGanttChart` – Calls the visualization for a passed text file (see example above).
- `mymake` – Translates and executes the different scheduling algorithms.
- `Task{.h, .cc}` – Contain the class `Task` that stores a task with all its properties (processing time, release time, deadline, etc.).

Here the classes `Task` and `Schedule` form an important basic component. With these classes, objects of the same type can be created by using their *constructors*. Furthermore, the classes contain a number of fields, which contain information belonging to the object. These fields cannot be accessed directly from outside, for example the main program, but only via certain access methods (set- and get-functions). The selected fields, constructors and methods are therefore now described in more detail.

### The class **Task**

- **Fields**

- `std::string name;` – Stores the name of a task as a text string.
- `unsigned int execTime;` – Stores the processing time of a task. Cannot be changed.
- `unsigned int remainingExecTime;` – Stores the remaining execution time of a task. Can be changed dynamically during program execution. Initially, the remaining execution time is equal to the execution time.
- `unsigned int releaseTime;` – Stores the release time of a task.
- `int deadline;` – Stores the deadline of a task. Default value is -1, this means the task has no deadline.
- `unsigned int period;` – Stores the period. Default value is 0, this means the task is not periodic.
- `int priority;` – Stores a static priority of a task. Default value is 0.
- `std::list<Task*> successorList;` – For tasks that have dependencies on each other. Stores a list of tasks that depend on the task as their direct successor.
- `std::list<Task*> predecessorList;` – For tasks that have dependencies. Stores a list of tasks that are direct predecessors of the task.

- **Constructors**

- `Task(std::string name, unsigned int execTime, unsigned int releaseTime);`  
– Creates an object with a name, an execution time, and a release time. Example:

```
Task* T1 = new Task("Frank", 20, 0);
```

A Task T1 with the name Frank, a processing time of 20 *time steps*<sup>1</sup> and a release time of 0 is created.

- `Task(std::string name, unsigned int execTime, unsigned int releaseTime, int deadline);` – As before, but in addition a deadline is passed.
- `Task(std::string name, unsigned int execTime, unsigned int releaseTime, int deadline, unsigned int period);` – As before, but additionally a period is passed.

## • Getters and Setters

- `std::string getName();` – Returns the task name as a text string. Example:

```
string s = T1->getName();  
cout << s << endl;
```

Assigns the name of task T1 to the string s and then displays it in the *Shell*.

- `unsigned int getExecutionTime();` – Returns the execution time.
- `unsigned int getRemainingExecutionTime();` – Returns the remaining execution time.
- `void setRemainingExecutionTime(unsigned int);` – Sets the remaining execution time. Example:

```
T1->setRemainingExecutionTime(8);
```

The remaining execution time of Task T1 is set to 8.

- `unsigned int getReleaseTime();` – Returns the release time.
- `void setReleaseTime(unsigned int);` – Sets the release time. This is useful for periodic tasks where the task reappears periodically.
- `int getDeadline();` – Returns the deadline.
- `void setDeadline(int);` – Sets the deadline. This is useful for periodic tasks where there is a different absolute deadline for each period.
- `unsigned int getPeriod();` – Returns the period duration.
- `int getPriority();` – Returns the priority.
- `void setPriority(int);` – Sets the priority.
- `void addSuccessor(Task*);` – Inserts a dependency between tasks. Example for two tasks T1 and T2, where T2 depends on T1:

```
T1->addSuccessor(T2);
```

- `std::list<Task*> getSuccessors();` – Returns a list with the direct successors of the task.
- `void addPredecessor(Task*);` – Inserts a dependency between tasks. Example for two tasks T1 and T2, where T2 depends on T1:

```
T2->addPredecessor(T1);
```

---

<sup>1</sup>The time is stored without unit, therefore it can be interpreted as *Time step, Cycle, Milliseconds, Seconds*, etc.



The methods `addSuccessor()` and `addPredecessor()` are redundant, the other variant serves only the comfort. Note: If a dependency is added to a task T1 using one of the two methods T2, the dependency is automatically added to task T2 in task T1 as well.

- `std::list<Task*> getPredecessors();` – Returns a list with the direct predecessors of the task.

## The class **Schedule**

### • Constructor

- `Schedule(Task* T, unsigned int duration);` – Creates a `Schedule` object for a given task T. At first an empty schedule with fixed length `duration` is created. Example:

```
Schedule* S = new Schedule(T1, 100);
```

A `Schedule` S for the task T1 is initialized for a length of 100 time steps.

### • Getters and Setters

- `void active(unsigned int);` – Sets the schedule to active at a time. Example:

```
S->active(4);
S->active(6);
```

This means that the task is executed at time steps 4 and 6.

- `void active(unsigned int, unsigned int);` – Sets the schedule for a time interval to active. Example:

```
S->active(2, 7);
```

This means that the task is executed in the time interval [2, 7).

- `bool isActive(unsigned int);` – Returns the value `true` (true) if the task is active at the given time, otherwise the value `false` (false).
- `void reset();` – Resets the complete schedule.
- `unsigned int getBeginTime();` – Returns the first time the task is active ( $\tau_b(T_i)$ ).
- `unsigned int getEndTime();` – Returns the last time the task is active ( $\tau_e(T_i)$ ).

### 3.1.1 How it works

In general, scheduling can be *event driven* or *time driven*. Depending on the scheduling variant, one or the other is advantageous. For the sake of uniformity, however, we will only use a time-driven simulation for a fixed duration of `SIMULATION_TIME` steps. Furthermore, individual scenarios are used (see directory `scenarios`), in which all *knowledge* (tasks with their properties) is already stored. A scenario is like a script that only the director knows, the audience only sees the individual actors when they appear on stage. Analogously, the simulation only sees the tasks when they are *released*.

**Task 3.1** Using the editor, open the source file `scenario1.cc` in the directory `scenarios`. For example with:

 `kate scenarios/scenario1.cc &`



View the contents of the file and determine which tasks with which properties are created. Fill in the following table, where  $d_i$  is the execution time,  $t_r(T_i)$  the release time and  $p_i$  the static priority of a task  $T_i$ :

|            | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | $T_6$ |
|------------|-------|-------|-------|-------|-------|-------|
| $d_i$      | 20    | 5     | 10    | 5     | 20    | 5     |
| $t_r(T_i)$ | 0     | 0     | 5     | 20    | 50    | 50    |
| $p_i$      | 2     | 1     | 4     | 3     | 6     | 7     |


Open the source file `scheduleFCFS.cc` in the directory `scheduler` with the editor. In this file the FCFS algorithm is implemented. Translate the program:

 `./mymake FCFS`

and visualize the schedule

 `./showGanttChart schedule.txt &`


Thoroughly study the source code of the FCFS scheduling algorithm (see Appendix B) and its functionality. If you have further questions, please ask your supervisor.

**Task 3.2** Calculate the individual and average flow and wait times of the tasks. To do this, open the source code of the main program `MainProgram.cc`. There are already suitable function trunks in the file. Complete them so that the desired times are calculated. 

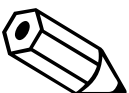
Translate and restart your program to verify your calculations ( `./mymake FCFS`).

## 3.2 Scheduling of Tasks without Deadlines


You have already become acquainted with the first algorithm for scheduling tasks without deadlines in the first two tasks, and further algorithms will be considered in the following tasks.

**Task 3.3** Open the source file for the scheduling algorithm SJF. The only difference to the FCFS procedure is the order in which the task queue is to be sorted. Complete the sort function `bool sortByExecutionTime(...)` at the beginning of the file accordingly. 

Translate and start ( `./mymake SJF`) your program and look at the schedule. Does it comply with the SJF algorithm?


**Task 3.4** Open the source file for the scheduling algorithm SRTN. Complete the sort function in an appropriate way. This algorithm allows the preemption of tasks. Analyze the differences between the implementation and the FCFS or SJF algorithms. 

Translate and run ( `./mymake SRTN`) your program and look at the schedule. Does it correspond to the SRTN algorithm?

**Task 3.5** Open the source file for the priority-based (PB) scheduling algorithm. Complete the sort function again appropriately. 

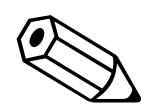
Translate and start ( `./mymake PB`) your program and look at the schedule. Does it correspond to the PB procedure?

**Task 3.6** Open the source file for the scheduling algorithm RR. Analyze how the *sequence* of tasks was implemented taking into account a given time quantum.

Translate and run ( ./mymake RR) your program and look at the schedule for a time quantum of 5, 3, 2 and 1.

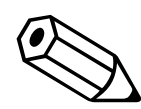
### 3.3 Scheduling of Tasks with Deadlines

In the following, we will look at scheduling algorithm for tasks with deadlines.



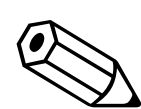
**Task 3.7** Scheduling with EDD. What must apply for release times for this to be applicable? Complete the file `scenario2.cc` according to the following table:

|            | $T_1$ | $T_2$ | $T_3$ | $T_4$ |
|------------|-------|-------|-------|-------|
| $d_i$      | 4     | 5     | 2     | 3     |
| $t_d(T_i)$ | 9     | 16    | 5     | 10    |



Open the source file for the scheduling procedure EDD. Complete the sort function in an appropriate way.

Translate and start ( ./mymake EDD) your program and look at the schedule. Does it meet your expectations? Are all deadlines met?



**Task 3.8** Next, the EDF scheduling algorithm is considered. Open the largely empty file `scheduleEDF.cc` and implement the EDF procedure. What is the best way to do this? Should you copy the EDD procedure, or which procedure is already available for implementation?

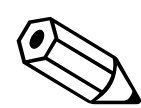
Translate and start ( ./mymake EDF1) your program and look at the schedule. Does it correspond to the EDF procedure?


**Task 3.9** Open the `scenario5.cc`, in which the tasks  $A$  to  $F$  are already defined as follows:

|       | A | B  | C | D | E  | F  |
|-------|---|----|---|---|----|----|
| $d$   | 2 | 2  | 2 | 2 | 2  | 2  |
| $t_d$ | 4 | 10 | 8 | 6 | 10 | 12 |
| $t_r$ | 0 | 1  | 1 | 3 | 7  | 9  |

Furthermore, the following dependencies are to apply for the tasks:  $A \rightarrow B$ ,  $A \rightarrow C$ ,  $B \rightarrow D$ ,  $B \rightarrow E$ ,  $C \rightarrow F$ . The dependence  $A \rightarrow B$  is implemented in (`scenario5.cc`) by the line

```
A->addSuccessor(B);
```



Add the other dependencies to the scenario. The defined task set should again be planned using the EDF procedure. Translate and start ( ./mymake EDF2) your program and look at the schedule. Are all deadlines and dependencies met?



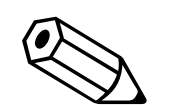
**Task 3.10** The same scenario (scenario5.cc) is now to be calculated with the scheduling algorithm EDF\*. The individual release times and deadlines must first be transformed according to their dependencies. Illustrate these transformations according to the lecture and exercise sheet 12. Open the source file scheduleEDFstar.cc. For the transformation of the release times, first a *topological sorting* is used [1], starting with tasks that do not depend on any other, from front to back, see functions bool topSortForward(...) and TQ.sort(topSortForward). Then the release times are transformed. The deadlines of the individual tasks can be transformed in a similar way. However, tasks that do not have successors are considered first, the topological sorting is done from back to front. Implement the function bool topSortBackward(...) and complete the calculation of the transformed deadlines. Translate and start (🔗 ./mymake EDF\_STAR) the program. Can all deadlines and dependencies be satisfied?

LARGE

### 3.4 Scheduling for Periodic Tasks

In the concluding part of this exercise, scheduling algorithms for periodic tasks are considered.

**Task 3.11** Open the file scenario6.cc and complete it for the following periodic tasks:



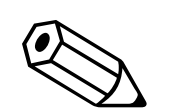
|          | $T_1$ | $T_2$ | $T_3$ |
|----------|-------|-------|-------|
| $d_i$    | 4     | 5     | 2     |
| $P(T_i)$ | 9     | 16    | 5     |

First, the periodic tasks are to be scheduled according to the EDF algorithm.

Translate and start (🔗 ./mymake EDF3) your program and look at the schedule, does it meet your expectations? Are all deadlines (periodically) met?

**Task 3.12** A modern aircraft is controlled by computers (*fly-by-wire*). For the control, the computers need various flight information, which must be determined and processed in different intervals: the acceleration values in x-, y- and z-direction every 5 ms, the three values of the rotation axes every 6 ms and the temperature every 10 ms. The processing of the acceleration values requires 1 ms, that of the rotations 2 ms and the temperature evaluation 3 ms. To ensure a trouble-free process, the acceleration values must already be calculated 5 ms, the rotary sensor values 4 ms and the temperatures 8 ms after the measurement.

Model the *Fly-by-Wire* scenario, in the file scenario7.cc according to your notes:



|              | $T_1$ (acceleration) | $T_2$ (rotation) | $T_3$ (temperature) |
|--------------|----------------------|------------------|---------------------|
| $d_i$        | 1                    | 2                | 3                   |
| $t_d^*(T_i)$ | 6                    | 6                | 11                  |
| $P(T_i)$     | 5                    | 6                | 10                  |

The scheduling should be done with the DM algorithm, Translate and start (🔗 ./mymake DM) the appropriate program and look at the schedule. Does it meet your expectations? Are all deadlines (periodic) met?



# A References

## References

- [1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 2001.
- [2] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., 2000.
- [3] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, 1987.
- [4] Jürgen Teich and Christian Haubelt. *Digitale Hardware/Software-Systeme: Synthese und Optimierung, 2. Auflage*. eXamen.press. Springer, 2007.
- [5] Übungsblätter zur Veranstaltung Eingebettete Systeme. [https://www.studon.fau.de/studon/goto.php?target=crs\\_5336524](https://www.studon.fau.de/studon/goto.php?target=crs_5336524).

## B Sourcefile `scheduleFCFS.cc`

```
bool sortByReleaseTime(const Task* Tl, const Task* Tr)
{ return (Tl->getReleaseTime() < Tr->getReleaseTime());
}

void scheduleFCFS(list<Task*> scenario)
{ // *****
  // * Create an empty schedule for all tasks in the scenario
  // *****
  map<Task*, Schedule*> taskSchedules;
  for (list<Task*>::const_iterator it = scenario.begin(); it != scenario.end(); ++it)
  { Task* T = *it;
    Schedule* S = new Schedule(T, SIMULATION_TIME);
    taskSchedules[T] = S;
  }
  // *****
  // * Create an empty task queue
  // *****
  list<Task*> TQ;
  // *****
  // * Main simulation loop
  // *****
  Task* T = NULL;
  list<Task*>::iterator it;
  for (unsigned int cycle=0; cycle<SIMULATION_TIME; cycle++)
  { // Update the task queue since new tasks might have been released
    updateTaskQueue(TQ, scenario, cycle);
    // Sort tasks according to their release times
    TQ.sort(sortByReleaseTime);

    // *****
    // * Implementation of FCFS scheduler
    // *****
    // Only if there are tasks in the queue or an active task,
    // we have something to do
    if ((!TQ.empty()) || (T!=NULL))
    { if (T == NULL)
      { // If there is no active task, dispatch the first task from the queue
        it = TQ.begin();
        T = *it;
      }

      // Schedule task in this cycle
      Schedule* S = taskSchedules[T];
      S->active(cycle);

      // Get remaining execution time of task T
      unsigned int d = T->getRemainingExecutionTime();
      if (d==1)
      { // The task has been totally executed and can be removed from the task queue
        TQ.erase(it);
        T = NULL;
      }
      else
      { // Decrement the remaining execution time by one cycle
        T->setRemainingExecutionTime(d-1);
      }
    }
  }
}
```

```
printSchedule(taskSchedules);  
generateGantt(taskSchedules, "schedule.txt");  
  
computeResponseTimes(taskSchedules);  
computeWaitingTimes(taskSchedules);  
}
```