IA158 - Scheduler

Jan Koniarik

February 20, 2023

Agenda

Project

- Your goal is to write a scheduler in C.
- It has to schedule multiple sets of tasks, two are specific:

Custom	A set of three predefined tasks, you have to add one task of your own. [1]
Sporadic	Two periodic tasks, and one special task for sporadic jobs.

- Your scheduler has to meet all the requirements, and you have three attempts at submission.
- You have to work alone.

[1] Please be creative

Scheduler assumptions

Assumptions:

- Jobs are non-preemptible
- Tasks are periodic
- One processor
- No resources/priorities/precedence
- Synchronized

Scheduler requirements

Requirements:

- Schedule our sets of tasks
- The schedule has to be valid
- The schedule shall not be hardcoded
- You have to use our clock API for time.

Git

During third or fourth week of the semester, you will get a repository for this course on `https://gitlab.fi.muni.cz`. We have access to the repository and we will deploy skeleton for the project there. You can't share this repository with anybody, except for the teachers of this course. It will serve as submission mechanism for the course.

Skeleton

There will be a simple C skeleton with CMake as a build system. The skeleton defines two components: scheduler library and demo application.
Only your library is evaluated and rated by us, we will link that library to our tests the same way it is linked to the demo application. That implies that if you break compability you fail the submission. [2]

[2] We suggest that you do not touch the CMakeLists.txt

Skeleton

Each set of tasks is defined in the skeleton, and schedulable with an algorithms based on the lecture. (But beware of your custom task!) Your solution has to schedule all sets of tasks correctly, in case there is an error - the project will not be accepted, and you fail.

Preemptability

But the non-preemptable scheduling is NP-hard.

We designed our tasks in a way that is solvable with an algorithm that expects preemptable jobs. It is your burden to design your task in a way, that it does not require preemptability.

Sporadic project server

One set of tasks has to support sporadic jobs. For that, you shall implement a periodic task representing a server. This server is specific to this project, lets call it a *project server*.
This server shall have abillity to queue up reasonable amount of jobs, and execute those once it's periodic job is executed. User of your schedule is expected to use ‘scheduler_on_sporadic‘ to insert a sporadic job into the system.

Sporadic project server

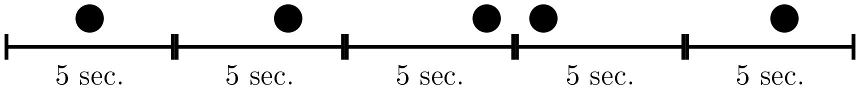
The implementation details of the server is up to you, only formal requirement is that the server is implement as any other periodic task. That is, there should be an instance of **struct task** that represent the server.
Apart from that, there are more requirements of what the server should be able to handle:

1. there should always be a capacity for at least one job appearing in any time during a 5s period
2. that job has relative deadline at least 2s
3. maximum execution time of said job is 20ms

The servers periodic job has to be configured in a way to meet these demands.

Sporadic job

The requirements of the sporadic job can be visualized in a following way:



You can see that during each 5s window, one job can appear at any time point. The jobs are represented with a black dots.

Sporadic job

There is *one job* in the formal requirements of this file. The project contains something that does not correlate to the requirements and may seem ... weird.
The key property of sporadic jobs is that the scheduler may deny them. The project intetionally generates more sporadic jobs than required by the formal requirements - so we can actually see the denial happening.

Implementation

- We will check your implementation of the scheduler - write it in a way that we can understand it.
    - Code quality (readability) is a necessity for a good scheduler, as the code has to be easy to understand and debug.
- You can not use dynamic memory.
    - Use fixed size buffers and fill them only partially.
    - Dyn. memory is not necessary in case the number of tasks is predictable and small.
    - The goal of the limitation is to constrain your creativity, there is simple solution to the project and this should partially prevent you from overengineering it.

API introduction

Interface

- File *task.h* contains a definition of the structure holding the tasks, job pointers, and sporadic tasks.
- $*$_*tasks.c/h* files provide a description of each set of tasks.
- *clock.h/c* contains API to work with time.
- *scheduler.h/c* contains API and structure for the scheduler that you have to implement.

Time limitations

To ease the development cycle you are developing on your OS. That is not real time and there is high chance you will notice the limits during development of the project.
Specifically, the functions designed for time management are not reliable. We decided to keep this solution that is simple, but not perfect.
We will tolerate any problems caused by the nature of OS, but it's your task to recognize that it is the OS, not your code that is the source of the problems.

Example task

Tasks

We will explain how this works on the first task set. We expect that you can handle the rest by yourself. These assigments are here for practice and are not part of the final project.

First set

The three provided tasks have these properties. All values are in milliseconds: [3]

led	period: 250, deadline: 50, max. exec. time: 1
uart	period: 251, deadline: 251, max. exec. time: 40
fib	period: 1499, deadline: 1499, max. exec. time: 40

The led task will be implemented as an example in this presentation.

[3]We will not change the values for tests. Hardcoded solutions for these numbers will be denied.

Assignment

1. Download project skeleton from git.
2. Check that you can compile it (find how to use CMake properly)
3. Open *main.c* file

Step 1: Write job function

The led task blinks a LED on imaginary embedded device.

```c
#include "task.h"

uint32_t i = 0;

void led_job(struct scheduler *, void *) {
    printf("Status of green led: %i", i);
    i = (i + 1) % 2;
}
```

Using a global variable is ugly, but we will live with that for now.

Step 2: Write an instantiation of task structure

We want the job of blinking LED to be executed at the 250ms period. This gives us 4 blinks per second. The maximum execution time is estimated at 1 ms. [4]

```
struct task LED_TASK_SIMPLE = {.period = 250,
                                .max_execution_time = 1,
                                .relative_deadline = 50,
                                .job = &led_job,
                                .data = nullptr};
```

[4] All time units are in milliseconds

Step 3: Write simple scheduler

As an example, we can show a simple execution of one task - a simple while loop. In the example, we use busy waiting to ensure the period the task has specified.

```
void schedule_single_task(struct scheduler *sched, struct
    task *task_ptr) {
    while (true) {
        uint32_t end_time = clock_time() + task_ptr->period;
        task_ptr->job(sched, task_ptr->data);
        clock_delay_ms(end_time - clock_time());
    }
}
```

Assignment

- Implement all three steps in previous slides.

More complex task

- We just made a really simple task with the scheduler capable of executing only that one task.
- This task only prints something based on the global variable.
- Usage of the global variable is not optimal - what if we would want to have multiple tasks with this job function?
  - That can be necessary for a lot of non-trivial tasks!
- We will fix that in the following modification!

Step 4: Data structure

The idea is to use different data for tasks with the same job function. For each task, remember a pointer for data and pass it to the function each time it is called. Given that we are working with C, we have to use void*.

```
struct led_task_data {
    uint8_t i;
};
struct led_task_data LED_DATA = {.i = 0};
struct task LED_TASK = {.period = 250,
                        .max_execution_time = 20,
                        .relative_deadline = 50,
                        .job = &led_job2,
                        .data = (void *)&LED_DATA};
```

Step 5: Modify function

Now, we can use that data structure in the function itself:

```
void led_job(struct scheduler *, void *void_data) {
    struct led_task_data *data = void_data;
    printf("Status of green led: %i", data->i);
    data->i = (data->i + 1) % 2;
}
```

Assignment

- Implement the fourth and fifth steps.
- Make a new instance of led task, with different data instances and same function.

Scheduler interface

The scheduler itself will have to use our API and data structure. See file *scheduler.h*. We use one instance of struct *scheduler* to represent the data of the scheduler. The definition of struct is empty, but you should fill it wisely for the scheduler. Periodic tasks call the function *scheduler_on_sporadic* to add a sporadic job into the scheduler structure. See the periodic tasks in sporadic set for details.
The function ‘scheduler‘ is the core function that does the scheduling. We expect that the function will never return. See ‘main‘ in demo app to understand how it is used.

Sporadic task

One set of tasks contains a sporadic jobs. These occur from within the other tasks at random moments. Remember to correctly deny the job in case you are not able to schedule it.

API Summary

Summary of the API:

- The basic unit is *struct task* contains:
  - timing constraints - period, relative deadline, and execution time
  - job function and job data
- API uses void pointer to pass data - you have to convert the pointer types manually
- Key function is ‘schedule‘ that executes the scheduling

Project

- Implement the scheduler for the task sets.
- There will be examination dates in the IS for project submission - you have to sign up.
  - Beware that the end of reservation date and submission date are apart from each other.
  - Once the submission date and time ends, we will pull the project from your repository for this course.
  - Only commits before the deadline will be taken into account.
  - In case you fail to submit the project, you lose one of the attempts for submission.

Communication

Prefered communication channels:

email	433337@mail.muni.cz

Or contact us in any other way you see fit. We will try to answer all of your questions, but we do not guarantee a fast response, and it may take me some time. Please be patient.

Experiment

As a little social experiment, please do send us an email once you finish *reading* this document. We promise that we won’t judge you for when that happens.