Task scheduler and parallel processing of multimedia data

The task scheduler should be implemented in an arbitrary object-oriented programming language. Define a programmatic API for the task scheduler and implement a logically separated GUI application for task scheduling that uses the defined API. Create a task type that performs parallel processing of multimedia data.

Task Scheduler

Properly implement basic functionalities for the task scheduler. The task scheduler must be implemented as part of a library. Provide a simple API that allows the user to schedule tasks. Allow the specification of the maximum number of tasks that can be executed concurrently. It is allowed to define your own API that must be used in the implementation of tasks, as a way to enable cooperative mechanisms with functionalities from the project task specification. It is also allowed to start the execution of each task as a separate process. Provide simple unit tests that demonstrate the task scheduler's operation.

The basic functionalities that the task scheduler API should provide are:

• Adding a task to the scheduler queue, with or without starting it

• Scheduling tasks while already scheduled tasks are being executed

• Starting, pausing, resuming, and stopping tasks

• Waiting for the end of task execution

• Specifying the scheduling algorithm, which includes FIFO (FCFS) and priority scheduling.

Optionally, tasks can be specified with: start date and time, total allowed execution time, and deadline for task completion or termination.

Parallel processing of multimedia data

Allow the specification of the allowed level of parallelism (e.g., allowed number of utilized CPU cores) for each task. Define a task type for multimedia file processing. Choose a specific algorithm for the course subject. Provide support for working with tasks of the defined type in the GUI application. The task of the defined type should be able to be scheduled for execution using the task scheduler. The task should allow the simultaneous processing of a large amount of input files.

Implement the task so that it can perform parallel processing on a single multimedia file, achieving acceleration (test with an adequately sized multimedia file).

Additional mechanisms for task scheduling

Allow for priority scheduling with preemption, as well as scheduling based on time slices, where tasks of higher priority receive more time slices for execution. Simulate preemption using cooperative scheduling mechanisms.

Enable working with resources, so that tasks can lock resources during execution, where resources are identified by a unique string. Provide mechanisms for preventing or detecting and resolving deadlock situations. Solve the problem of priority inversion (PIP or PCP algorithm).

GUI application and task persistence

Provide an application with a non-blocking graphical user interface for task scheduling. Construct the application in a way that new types of tasks can be easily added. Allow the user of the application to specify the scheduling method. Allow the user of the application to specify all properties of the created task (allowed execution time, deadlines, etc.). The GUI application should support the viewing of tasks in progress, with progress bars reporting the progress of the task, as well as the creation, starting, pausing, resuming, stopping, and removing of completed or stopped tasks. Enable task serialization and deserialization. Serialized fields may include, for example, an array of input file paths and a path for generating output files.

User-space file system

Write a driver for a user-space file system. The file system should use the scheduler API and allow for the processing of multimedia data by tasks in the following way:

The file system should contain an input folder into which multimedia files can be copied.

After the files are copied into the input folder, the processing of those files begins.

After the processing of the files is complete, the results should be available in the output folder.