Real-Time Scheduling Simulator

Description

This project demonstrates the fundamentals of real-time scheduling. A real-time system consists of a set of real-time tasks scheduled to run upon a platform of processor(s). The task model considered in this project is briefly described below.

Task System Model

A task τ_i is characterized by a tuple (C_i, D_i, T_i), where C_i is the worst-case execution time (WCET), D_i is the relative deadline, and T_i is the period or minimum inter-arrival time between two consecutive releases of the task. We consider an implicit-deadline task-model such that $D_i = T_i$. A task τ_i generates a potentially infinite number of successive releases (or jobs). The utilization $U_i = C_i/T_i$ is the processor demand of the task.

In addition to the static parameters of a task, the dynamic attributes during the run-time at timestep t include (i) the pending execution time $C_i(t)$, (ii) the residual relative deadline $D_i(t)$, and (iii) the instantaneous laxity $L_i(t) = D_i(t) - C_i(t)$.

Given the task model described above, the task system comprises a set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n independent implicit-dealine tasks to execute on a homogeneous platform of m cores (or processors) (i.e., all cores have the same computing capabilities and are interchangeable), following a preemptive global scheduler. The total utilization of a task set must be less than or equal to the total capacity of the processor platform. In this project, we consider unit-speed cores, therefore $U = \sum_{i=1}^{n} U_i \leq m$

Preemptive Global Schedulers

A preemptive scheduler allows the executin of a task's job to be preempted by the job of another task (usually of higher priority) to ensure the timing requirements are met (in this, tasks do not miss their deadlines). This project implements the following preemptive global schedulers:

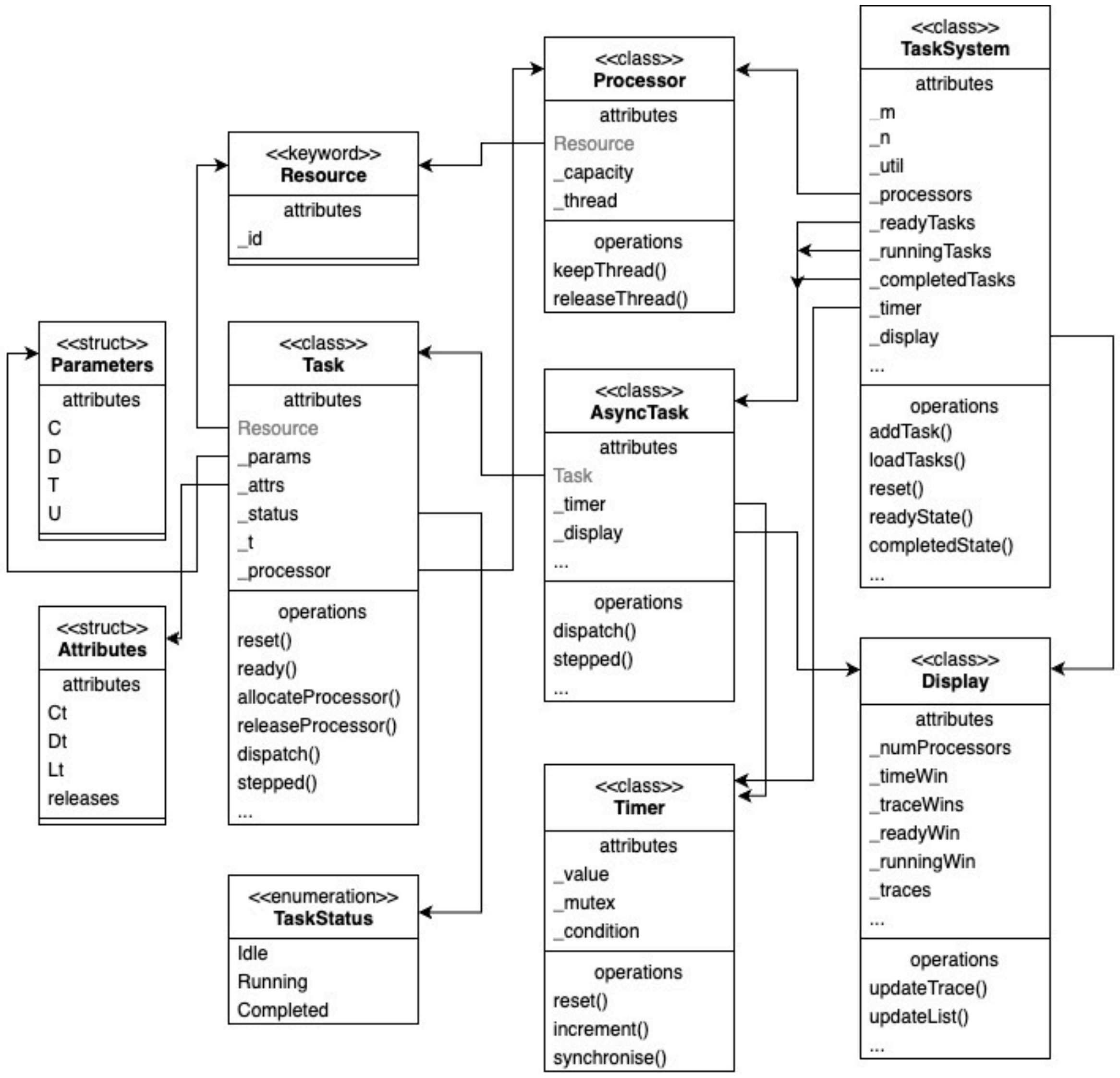
- Proportional Fairness (pFair)
- Least Laxity First (LLF)
- Deadline Monotonic (DM)
- Earliest Deadline First (EDF)

File and Class Structure

File Structure

```
.
├── build               # Compiled files
├── include             # Header files
│   ├── algorithms      # Header files for scheduler
algorithms
│   │   ├── PFair.hpp
│   │   └── PriorityDriven.hpp
│   ├── AsyncTask.hpp
│   ├── Display.hpp
│   ├── Processor.hpp
│   ├── Resource.hpp
│   ├── Task.hpp
│   ├── TaskSystem.hpp
│   ├── Timer.hpp
│   └── utils.hpp
├── src                 # Source files
│   ├── algorithms      # Implementation of scheduler
algorithms
│   │   ├── PFair.cpp
│   │   └── PriorityDriven.cpp
│   ├── AsyncTask.cpp
│   ├── Display.cpp
│   ├── main.cpp
│   ├── Processor.cpp
│   ├── Resource.cpp
│   ├── Task.cpp
│   ├── TaskSystem.cpp
│   └── Timer.cpp
├── tasksets            # Taskset files
│   ├── example.txt
│   └── ...
├── LICENSE
└── README.md
```

Class Structure



Building and Running

The following steps are required to build and run the project:

1. Install NCURSES library. Although the library is usually shipped with C++ installation, in case the library is not already installed or want to compile from source:
   - download the package from ftp://ftp.gnu.org/pub/gnu/ncurses/ncurses.tar.gz
   - read the README and INSTALL files for details on to how to install it. It usually involves the following operations. (ref: https://tldp.org/HOWTO/NCURSES-Programming-HOWTO/intro.html)
2. Create a build directory and build with CMAKE.
3. Run the executable with the following options (command line arguments):
   - `./RTSSimulator <TASKSET_FILENAME> <NUM_PROCESSORS> <NUM_STEPS> <SCHEDULER>`
   - e.g. `./RTSSimulator tasksets/example.txt 2 1000 pFair`
   - `<SCHEDULER>` option can be any of `pFair, LLF, DM, or EDF`.
4. Check the examples tasksets provided in `tasksets/example.txt` and `tasksets/canonical.txt`. The structure is as follows:

```
<TOTAL_UTILIZATION>
<NUM_TASKS>
<$C_1$, $T_1$>
<$C_2$, $T_2$>
...
```

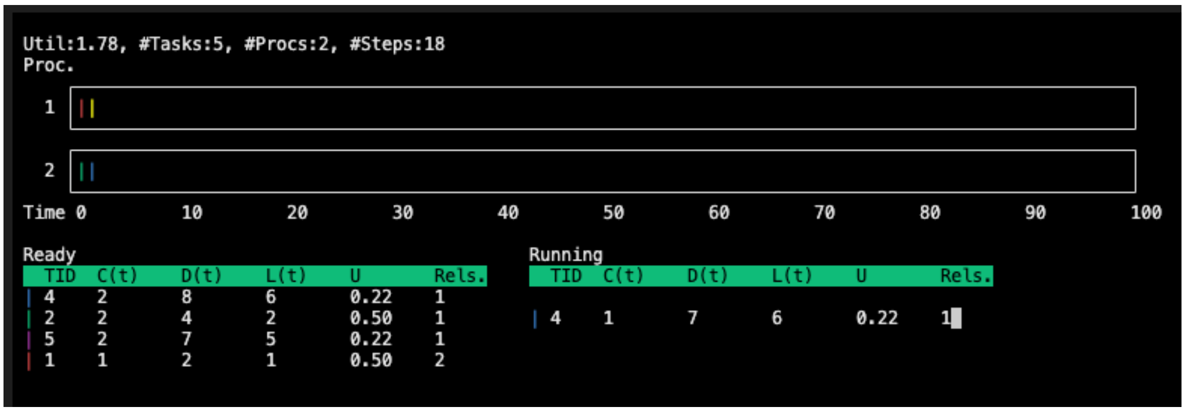
The Expected Behavior

The RTSSimulator executor is run, specifying the taskset file, the number of processors, the number of steps, and the scheduling algorithm as described above. pFair is the default scheduler and the default number of time_steps is the lowest common multiple (LCM) of the tasks' periods (T's).

When the program is launched, a `TaskSystem` instance is created with the specified number of processors. Then the task system loads the tasks from the input file and validates that the sum total utilization of the tasks does not exceed the capacity of the cores.

The scheduling algorithm queries the ready (and completed if necessart) state(s) of the task system, orders the jobs of the tasks in the ready 'queue', and outputs the indices of the top m ready jobs. These indices are used by the task system to run the corresponding jobs each of which is allocated a processor on which to run. Theremaining ready and completed jobs are idled. All the jobs are dispatched to step for the specified time delta. When a job completes its execution, it is moved to the completed tasks set, waiting for its next job release when it is then moved to the ready 'queue'.

The traces of the execution of jobs are shown through the NCURSES display and the dynamics of the ready and running jobs are listed. An example output is provided below:



```
Util:1.78, #Tasks:5, #Procs:2, #Steps:18
Proc.
  1 ||
  2 ||
Time 0        10        20        30        40        50        60        70        80        90        100

Ready                                               Running
 TID  C(t)   D(t)   L(t)   U      Rels.              TID  C(t)   D(t)   L(t)   U      Rels.
| 4   2      8      6      0.22   1
| 2   2      4      2      0.50   1                 | 4   1      7      6      0.22   1
| 5   2      7      5      0.22   1
| 1   1      2      1      0.50   2
```

Addressed Rubric Points

Loops, Functions, I/O

1. The project demonstrates an understanding of C++ functions and control structures.

   - A variety of control structures are used in the project: Control structures are used throughout the project. Examples of interesting ones are found in `src/main.cpp lines 34-51` and `src/TaskSystem.cpp lines 222`

`- 243`.

- The project code is clearly organized into functions: Object Oriented Programming is adopted and the complex program is divided into components each of which groups related functions.

2. The project reads data from a file and process the data, or the program writes data to a file.

    - The project reads data from an external file or writes data to a file as part of the necessary operation of the program: The program takes inputs as command line arguments when the program is run (see `src/main.cpp lines 15-24`). Additionally, one of the inputs is a filename to load the tasks from disk (see `src/TaskSystem.cpp lines 158-173`).

3. The project accepts user input and processes the input.

    - The project accepts input from a user as part of the necessary operation of the program: The program takes inputs as command line arguments when the program is run (see `src/main.cpp lines 15-24`)

Object Oriented Programming

1. The project uses Object Oriented Programming techniques.

    - The project code is organized into classes with class attributes to hold the data, and class methods to perform tasks: The project heavily uses OOP by dividing up the operations and grouping related logics and data into appropriate classes.

2. Classes use appropriate access specifiers for class members.

    - All class data members are explicitly specified as public, protected, or private: Done!

3. Class constructors utilize member initialization lists.

    - All class members that are set to argument values are initialized through member initialization lists: See `src/TaskSystem.cpp line 14`, `src/AsyncTask.cpp lines 5-10`, `src/Display line 3`, `src/Processor.cpp line 3`, and `src/Task.cpp lines 5-6`.

4. Classes abstract implementation details from their interfaces.

    - All class member functions document their effects, either through function names, comments, or formal documentation. Member functions do not change program state in undocumented ways: All the class member functions use clear function names, comments, and in some cases doumentation to describe their effects. See `src/TaskSystem.cpp lines 76, 112, 175, 209` for examples.

5. Classes follow an appropriate inheritance hierarchy.

    - Inheritance hierarchies are logical. Composition is used instead of inheritance when appropriate. Abstract classes are composed of pure virtual functions. Override functions are specified: See `include/Task.hpp line 11` and `include/AsyncTask.hpp line 11` for inheritance examples; `include/TaskSystem.hpp lines 52-53` for composition examples; and `include/Task.hpp lines 60-61` and `include/AsyncTask.hpp lines 21-22` for virtual and override examples.

6. Overloaded functions allow the same function to operate on different parameters.

    - One function is overloaded with different signatures for the same function name: See `include/TaskSystem.hpp lines 57-58`.

7. Derived class functions override virtual base class functions.

    - One member function in an inherited class overrides a virtual base class member function: See `include/Task.hpp lines 60-61` and `include/AsyncTask.hpp lines 21-22`.

8. Templates generalize functions in the project.

    - One function is declared with a template that allows it to accept a generic parameter: See `include/utils.hpp lines 11 and 21`.

Memory Management

1. The project makes use of references in function declarations.

    - At least two variables are defined as references, or two functions use pass-by-reference in the project code: See `include/TaskSystem.hpp lines 56-59`.

2. The project uses destructors appropriately.

    - At least one class that uses unmanaged dynamically allocated memory, along with any class that otherwise needs to modify state upon the termination of an object, uses a destructor: See `src/Display.cpp line 21` and `src/Processor.cpp line 24`.

3. The project uses scope / Resource Acquisition Is Initialization (RAII) where appropriate.

    - The project follows the Resource Acquisition Is Initialization pattern where appropriate, by allocating objects at compile-time, initializing objects when they are declared, and utilizing scope to ensure their automatic destruction: See `src/TaskSystem.cpp lines 20-21`.

4. The project follows the Rule of 5.

    - For all classes, if any one of the copy constructor, copy assignment operator, move constructor, move assignment operator, and destructor are defined, then all of these functions are defined: See `include/TaskSystem.hpp lines 20-24`, `include/AsyncTask.hpp lines 15-19` and `include/TaskSystem.hpp lines 20-24` for examples.

5. The project uses move semantics to move data, instead of copying it, where possible.

    - For classes with move constructors, the project returns objects of that class by value, and relies on the move constructor, instead of copying the object: See `src/main.cpp lines 37-39`.

6. The project uses smart pointers instead of raw pointers.

    - The project uses at least one smart pointer: unique_ptr, shared_ptr, or weak_ptr. The project does not use raw pointers: See `include/TaskSystem.hpp lines 14, 52-53` for examples.

Concurrency

1. The project uses multithreading.

    - The project uses multiple threads in the execution: See `include/Processor.hpp lines 17, 22` and `src/AsyncTask.cpp line 51`.

2. A promise and future is used in the project.

    - A promise and future is used to pass data from a worker thread to a parent thread in the project code: Not used!

3. A mutex or lock is used in the project.

    - A mutex or lock (e.g. std::lock_guard or `std::unique_lock) is used to protect data that is shared across multiple threads in the project code: See `src/Timer.cpp lines 4, 10, and 15` for examples.

4. A condition variable is used in the project.

    - A std::condition_variable is used in the project code to synchronize thread execution: See `src/Timer.cpp lines 6 and 11` for an example.