rtsim: a real-time scheduling emulator
Benjamin R. Straw - EEE 4775 Fall 2020

Abstract
The education of real-time scheduling topics often includes the discussion of complex scheduling systems and algorithms. Among the course materials utilized, there often is a need for a series of examples for each represented algorithm, both of valid schedules and task sets which do not meet schedulability requirements, allowing students the opportunity to visualize a series of output schedules relating to a specific algorithm and gain more insight into the specific operation of the algorithm.

Presented in this paper is a real-time scheduling framework which, when given an input series of tasks and a duration of time through which to simulate, creates a record of all of the events on a hypothetical uniprocessor with no preemption penalties. This mirrors the uniprocessor model studied when dynamic- and static-priority algorithms are first introduced to a student. Implemented using modern C++14 programming practices and designed with extensibility as a core consideration, the software welcomes users to represent additional scheduling algorithms or produce different outputs to meet the needs of their specific situation.

Design
While the software referenced in this paper does execute as a standalone program on any modern operating system, the existing main.cpp file serves primarily as an example utilization of what can best be described as a programming library: rtsim provides an abstract Scheduler class which is extended with the additional functionality of a specific scheduling algorithm.

A short example of the usage of this framework and the power of this extensible design is included below as listing 1, where a sample task set is initialized and provided as input to a deadline-monotonic scheduler. The output of this program is an accurate representation of the state of the processor at each point of the schedule, checked against the provided timing diagram in the chapter 3 slides from which the task set is derived. This example could be modified by using a different scheduler (the other two provided implementations are EDF and NP-EDF).

```cpp
#include <iostream>
#include "DM.h"

int main() {
    taskset_t t1 = {
        // Taskset from chapter 3
        // slides, page 17
        // (proof of non-tightness).
        task_t {
            .period = sclk::ms(20),
            .cost = sclk::ms(6),
            .deadline = sclk::ms(10)
        },
        task_t {
            .period = sclk::ms(50),
            .cost = sclk::ms(23)
        },
    };

    DM sim(t1, sclk::ms(12));
    sim.run();
    sim.print_sched();

    return EXIT_SUCCESS;
}
```

Listing 1: Demonstration of framework usage

As demonstrated, the framework is quite self-contained, and could be integrated into existing designs with minimal effort. Once a task set has been created and a scheduler
initialized, a call to .run() performs all of the simulation work in a single blocking call. The subsequent .print_sched() simply outputs the internal event vector to the standard output of the program, though additional outputs could be implemented (this will be discussed in the Future Considerations section).

Custom Clock

The system at the core of this framework is the custom clock module. The C++11 standard revision introduced a new branch of the standard library, the chrono namespace, defining new chronology concepts such as durations and time points, which themselves are dependent on a clock implementation.[2] While most pieces of software developed utilizing these new chronology features make use of the provided clock implementations (most commonly the monotonic-guaranteed steady_clock or the system-dependant high_resolution_clock), these clock implementations are driven by the operating system and can not be controlled by the executing program. For the purposes of rtsim, a different mechanism was required, one which would allow the program to manage the clock time manually and only progress time at certain intervals; however, for sake of conformance to modern C++ standards, use of the chronology features provided by the system was preferred to implementing a new and unique time-keeping system.

C++11 defined a new named requirement, TrivialClock[3], which is a superset of the requirements to satisfy Clock[4]. These requirements dictate what functionality must be present for a clock implementation to interoperate cleanly with the chrono standard library. A class sclk (short for simulation clock) is implemented in the files sim_clock.h and sim_clock.cpp. The class itself is quite simple, defining the most basic class structure and providing the most basic implementations of the clock functionality. Critically, the clock does not convey any real-world or system timing information; instead, the clock is ticked manually by the program.

The first two lines of the class definition provides two of the most important pieces of information about the clock. First, that the internal representation of time is done with an unsigned 64-bit integer. The choice of integer over a floating-point unit is significant here; the C++ chrono standard provides for both integer and floating-point representations of time as the unit underlying a clock, however for simulation purposes it was deemed essential to have an indivisible unit of time to represent an execution cycle, a sort of quantum of time. To allow a base time unit to be infinitely divisible would complicate the process by which time is advanced in the simulation, as well as potentially cause timing inaccuracies over large simulation periods. The second line of the class definition reveals that the base unit of time is a microsecond. Using the same reasoning as above, a single tick of the clock system is intended to represent a single unit of execution on the simulated processor, where microseconds are assumed to be of significant granularity to represent low-power systems.

Additionally provided in the class definition are some local definitions of common durations, microseconds, milliseconds, and seconds. These definitions limit the amount of extraneous typing required and reduce error.

Task System

With the custom clock system in place, a task system can be implemented. We first break down the definition of a task, provided for reference in listing 2.
The task definition follows very closely to the academic definition of a task: a period, an execution cost, a deadline, and a phase. Additionally included are a priority which may be provided initially by the user for some scheduling implementations and is used internally by others, a task ID which is generated internally and used to refer to a task across the run, and a boolean flag demarking whether the task should repeat. It is worth noting that the only values required to be populated at initialization are the period and the execution cost, with all other values being set to sane defaults.

It is important to remember when reviewing the structure of a task that `sclk::duration` represents a span of time on the simulation clock. The period, execution cost, deadline, and phase are all represented as these durations, not as single points of time. Thus, the deadline is relative to the start of each job of the task, and the phase is the offset from the zeroth tick of the clock that the first job of the task will arrive. Likewise, `prio_t` is a type definition aliased to `unit16_t`, such that priorities in the system are represented with an unsigned 16-bit integer.

Complementary to tasks are jobs, which are also represented in the system. The definition of a job in rtsim is provided in listing 3.

Unlike tasks, which operate almost exclusively with durations, a job has a specific time of arrival and a specific deadline. These are calculated from the underlying task when the job is created. Likewise, the execution cost remaining on a job is initially set equal to the underlying task’s execution cost, however as processor time is allocated to a job the performed work is subtracted from its cost remaining. Each job is assigned a job ID, beginning at zero for the first job of a task and incremented by one for every subsequent job. Job IDs are sequential and shared; to uniquely identify a specific job, a (task ID, job ID) pair is required. The reference to the underlying task can be utilized to retrieve task information for this purpose and others, such as checking priority. Finally, a boolean flag demarks whether the task has been completed, which is used when monitoring for missed deadlines.

**Driver**

The primary execution loop of rtsim is the driver. This is implemented in C++ as an abstract base class from which each scheduling algorithm implementation is derived, and it provides to each implementation a consistent set of requirements.

When rtsim is used from a main program, it is via the instantiation of a derived scheduling algorithm implementation. However, this derived class is required to instantiate the base class, which performs the simulation setup. The
driver receives from the derived class (which itself receives from the user program) a task set and a duration to simulate for, relative to the zeroth simulation tick. In the constructor of the driver, the tasks in the input set are provided unique task IDs, and the first job for each task is created and added to the arrival queue. These first jobs are all assumed to arrive at the zeroth tick of the simulation unless a phase is provided in the task definition. Once this constructor is complete, it returns up to the derived class to perform any construction tasks it may require.

The user program which instantiates the scheduler object is responsible for calling the .run() function, a blocking call which performs the entire simulation. This function returns a boolean value representing the validity of the schedule; a return value of true means the input task set was scheduled with no missed deadlines prior to the end of the simulation period. It is worth noting that the input task set could still result in a missed deadline; a true result is only valid from the zeroth simulation tick until the provided simulation duration is reached.

For each clock tick from zero until the duration has been reached, the .run() function first iterates over every job in the arrival queue. The arrival queue is a vector of job instances first populated from the initial task set in the constructor; on every simulation loop any job which has an arrival time equal to the current simulation tick is added to the vector of active jobs and is also announced to the scheduling algorithm. Prior to a jobs arrival time, the scheduling algorithm is unaware of its existence. For every task that is not configured as a single-run task, once a pending job has arrived and is sent to the scheduling algorithm, another instance of the job is created with an incremented job ID and an arrival time of the current simulation tick plus the task period.

After the newly arrived jobs have been announced to the scheduling algorithm, the process step begins. This step varies for each algorithm; the driver defines a function signature for process() that is pure virtual and must be overridden by the derived class. Once the scheduling algorithm returns from process(), the simulation clock is advanced one tick, all of the incomplete jobs are checked for missed deadlines, and the simulation begins another iteration of the loop.

Throughout the simulation, the driver and the scheduler implementation both utilize an event log referenced in the code as jsched. This represents the events that occur on the simulated uniprocessor, stored in an Event type as described in listing 4. While the simulation is taking place, the vector of events jsched is continuously appended to. Once the simulation has completed execution, the events can be retrieved and used to build an output from rtsim. The code supplied with this report only supports plain-text output to the standard library of each event and all the relevant information about it, but additional uses of this event information are explored in the Future Considerations section.

```cpp
enum Types {
    IDLE,
    ARRIVAL,
    START,
    PREEMPT,
    COMPLETE,
    MISS
};

struct Event {
    sclk::time_point time;
    Types type;
    std::shared_ptr<job_t> job;
    sclk::duration remaining;
};
```

Listing 4: The Event definition
**EDF Implementation**

With the three core systems of rtsim described, the clock, the tasks, and the driver, we now turn our attention to the specific implementations of the three provided scheduling algorithms. EDF is covered first, as the transition from EDF to NP-EDF is then trivial.

The most important component of the EDF implementation is the data structure which holds any newly-arrived jobs announced to the implementation: a multimap. An associative container from the standard library, the multimap is a key-value container where the elements are stored in order, sorted by keys, and duplicate keys are allowed. The EDF implementation utilises these properties in a map where the key is a job’s deadline and the value is a pointer to that job. Since duplicate keys are supported, so too are multiple jobs with the same deadline. The ordered property of the multimap ensures that the element stored at the front of the data structure will always be the job with the earliest deadline; this is possible since the deadlines are stored as time_points of the sclk, which adheres to the TrivialClock requirement of the chrono library and is thus natively understood by the rest of the standard library. This multimap is referred to within the EDF implementation as the edf_queue.

When EDF’s `process()` function is called, the implementation first checks whether the edf_queue is empty; if the queue is empty and there is no active job running according to the driver, we are in a period of processor idle time, which is accordingly added to the event log. Otherwise, the job which is to be executed is selected. This job is always the job at the front of the edf_queue, obtained using the `begin()` function. The `process()` function then checks whether the job selected for execution is the same as the job most recently reported as executing to the driver; if not, a preemption event is recorded to the event log, followed by a job execution event. Finally, a clock tick worth of time is removed from the selected job’s execution time remaining, and the job is checked for completion; if the job has no more execution time remaining, the job is marked as complete, a job completion is reported to the event log, and the job is removed from the edf_queue.

**NP-EDF Implementation**

The non-preemptive variant of EDF is implemented in much the same way as the preemptive EDF, and uses much of the same code. The NP-EDF implementation utilizes the same multimap data structure, with a job’s deadline as the key and a pointer to the job as the value; any newly-arrived job announced to the implementation by the driver is added to the multimap in the same was as with EDF. The primary difference between the two lies in the job selection step: whereas EDF always selects the job for execution as the element currently at the front of the multimap edf_queue, the NP-EDF implementation only does this if there is not a job currently running; a job in active execution is always selected for continued execution regardless of the state of the edf_queue, since jobs can not be preempted in NP-EDF.

**DM Implementation**

The final scheduling algorithm provided by rtsim is a static-priority deadline-monotonic scheduler. Instead of an ordered map to organize the jobs that have arrived to the scheduler but have not yet been completed, the DM implementation uses a vector of queues, the dm_queue.

When the DM scheduler is constructed, the input task set is sorted by relative deadlines from shortest to longest. The tasks are then re-assigned task IDs as well as priorities based on this order, with the shortest relative deadline...
receiving the first task ID and priority. The DM constructor also expands the vector \( \text{dm}_\text{queue} \) to contain as many empty queues as there are tasks in the input task set.

When a newly-arrived task is received from the driver, the DM scheduler inspects the new job’s underlying task to retrieve the task’s priority \( n \). This priority \( n \) is used as an index into the \( \text{dm}_\text{queue} \) vector to retrieve the \( n \)-th queue. Under this system, all jobs of the single task with priority 2, for example, are guaranteed to use the queue located at \( \text{dm}_\text{queue}[2] \).

When the DM implementation’s \( \text{process()} \) function is called, the vector \( \text{dm}_\text{queue} \) is iterated over, searching for a non-empty queue. This iteration occurs in order, first checking the queue at \( \text{dm}_\text{queue}[0] \), then the queue at \( \text{dm}_\text{queue}[1] \), and so on until the end of the vector. If each queue within the \( \text{dm}_\text{queue} \) vector is empty, no job is executed and the simulated uniprocessor is in an idle period. If any non-empty queue is encountered in the vector, the job at the front of this queue is selected for execution. Under this system, the system is guaranteed to always select for execution the first-available job with the highest priority.

Implementations Summary

Each provided implementation follows largely the same structure: they all of have the same checking for an idle period, the selection of a job to be executed, the reporting of that execution back to the driver’s event log, the subtraction of a processor tick from the job’s execution cost remaining, and then a check for job completion. These elements, described in detail in the EDF Implementation section, are also present in the other two implementations with minor adjustments as necessary to account for data structure differences between the implementations. The big difference between implementations is the job selection process and their underlying data structures in which they store their pending jobs.

Future Considerations

The code described in this paper serves primarily as a proof-of-concept, performing the schedule creation task with little additional functionality. Three main considerations for future extension to this software include improving the input interface, the output options, as well as the number of schedulers available.

The software in its current state has no user-friendly interface: a task set must be created by hand and a scheduler instantiated with the task set and simulation duration passed directly. A more friendly solution would be to accept command-line arguments indicating a file path to read a task set from, as well as the simulation duration and which scheduling algorithm to use.

Similarly, while the software does currently print a human-readable schedule upon simulation completion, additional output options may be desired. The simulation process actually occurs in an output-agnostic manner: the scheduler implementations report events to the driver’s event log, and only after completion of the simulation is the event log iterated and output. In this system, it is trivial to add additional output methods, such as a machine-readable CSV file.

Finally, additional scheduler implementations are not only possible but encouraged. As described in the Implementations Summary section, under most circumstances only the job selection and pending-job storage structures will need significant modification. This allows for the easy extension by educators to provide examples of new scheduling algorithms, or as possible assignments for students.
References

[1] EEE 4775 Fall 2020 slides “originated from Prof. J. H. Anderson at UNC-CH” and provided by Prof. Zhishan Guo