COMP 737 Project - Schedule Simulator

Eli Zachary

1 Introduction

In class, we regularly discussed cheap, efficient ways to check the schedulability of task sets under a given scheduling algorithm. However, in the real world, these algorithms must be actually implemented within a real-time system in order to put the theory into practice. The main product of this project is my attempt at doing so. It is a computer application which determines the schedulability of real-time task sets according to various scheduling algorithms. The application determines schedulability not through mathematical scheduling tests, but by simulating each time instant at which a job in the task set is released, completed, or has its deadline, and checking whether any deadlines have been missed. The simulator is capable of processing seven different scheduling algorithms, which include both uniprocessor and multiprocessor algorithms. It is programmed in Python3.

2 Data structures

In order to represent real-time systems theory in code, several RTS concepts must be converted into data structures which Python can work with. Before looking at the actual code, we must give an overview of these data structures.

2.1 Tasks and task sets

Each task $\boldsymbol{\tau}_i$ in a task set $\boldsymbol{\tau}$ is defined with the following parameters: phase ϕ_i, period T_i, execution cost C_i, relative deadline D_i, and a unique task ID I_i. The first four parameters are expressed in unspecified time units. Formally:

$$i \in \mathbb{N}_0, \boldsymbol{\tau}_i \in \mathbb{N}^5, \boldsymbol{\tau}_i = (\phi_i, T_i, C_i, D_i, I_i)$$

$$\phi_i \in \mathbb{N}_0, T_i \in \mathbb{N}_1, C_i \in \mathbb{N}_1, D_i \in \mathbb{N}_1$$

A task set is represented by an array of instances of our task data structure. The task set array is unordered, as most of the simulator's sorting requirements come from individual jobs rather than tasks. The simulator is capable of simulating any system of tasks regardless of synchronicity or whether D_i is less than or greater than P_i. This includes task sets which have context-switching costs, but not those that require a model of constraining access to resources. Nor does it support scheduling aperiodic tasks.

One important limitation of the schedule simulator's implementation is that all parameters must be natural numbers. This is to avoid headaches involving Python's representation of floating point numbers.

2.2 Jobs

Each instance of a job $J_{i,n}$ is defined with the following parameters: parent task ID I_i, remaining time until absolute deadline $RD_{i,n}$, and remaining execution time $RC_{i,n}$. The last two parameters are expressed in unspecified time units. Formally:

$$i \in \mathbb{N}_0, n \in \mathbb{N}_0, J_{i,n} \in \mathbb{N}^3, J_{i,n} = (I_i, RD_{i,n}, RC_{i,n})$$
$$I_i \in \mathbb{N}_0, RD_{i,n} \in \mathbb{N}_0, RC_{i,n} \in \mathbb{N}_0$$

Once again, all of these parameters must be natural numbers. Note that these are not the typical parameters for representing a job instance. In this simulation, each individual job instance keeps track of the time remaining until its absolute deadline, and its remaining execution time. At the instant the job instance is released, $RD_{i,n}$ is equal to its parent task's relative deadline, and $RC_{i,n}$ is equal to its parent task's execution cost. $RD_{i,n}$ decreases by one for every one time unit that passes in the simulation, while $RC_{i,n}$ decreases by one for every one time unit in which $J_{i,n}$ is active. This model allows us to better handle cases where there might be more than one instance of a job released at any given time, such as when $P_i < D_i$, for example.

2.3 Context switches

For any given run of the simulator, there is one universal context-switching cost $csc \in \mathbb{N}$ (by default set to 0), which can be set by the user. If $csc > 0$, the simulator will use a special context switch object CS_j. This object, similar in structure to a job, is used to represent time spent context-switching into or out of a job. CS_j is defined by its parent task ID I_i, IO_j, a Boolean value that returns true if CS_j represents a context switch into a job and returns false if CS_j represents a context switch out of a job, and its remaining context-switching time RC_j. It also optionally contains L_j, a link to another object. If CS_j is a context switch into a job $J_{n,i}$, $L_j = J_{n,i}$. If CS_J is a context switch out of a job that was preempted by another job, $L_j = CS_{j+1}$, with CS_{j+1} being a context switch into the new job. L_j is null if CS_j is a context switch out of a job which is not being preempted. Formally:

$$n \in \mathbb{N}_0, CS_n = (I_i, IO_j, RC_j, L_j)$$
$$I_i \in \mathbb{N}_0, IO_j \in \{0,1\}, RC_j \in \mathbb{N}_0, L_j \in \{J, CS, \emptyset\}$$

Context switches can, in some cases, be treated as a job. Similarly to $RD_{i,n}$ with jobs, RC_j is initially set to csc and decreases by one per every one time unit that passes in the simulation. The most important differences between context switches and jobs are that context switches do not have a deadline (and therefore cannot fail to meet a deadline), and that they are non-preemptive regardless of the scheduling algorithm being used.

On many scheduling algorithms, the amount of extra time taken up by a job's context-switching costs can easily be determined with an equation: $C_i + csc = C_i + 2(K_i + 1)$ [2], rather than adding an additional structure with some unique behaviors like this implementation does. Why not simply add $2csc$ to a job's execution cost every time a context switch occurs? Context-switching time needs to be handled separately because, if additional context-switch time were added to a job's normal execution time, the job could then be preempted by another job during the time that represents context-switching. Additionally, time spent context-switching out of a job can occur after a job's deadline, which would be much more difficult to represent if context-switch time was simply added to execution cost.

2.4 Active and queued job arrays

Released jobs are stored one of two locations: the active jobs array or the queued jobs array.

The active jobs array's size is equal to the number of processors, as it essentially represents the processors themselves. Jobs in the active jobs array will execute until either the job is finished or it is preempted by a higher-priority job. A job may never move from one position in the active jobs array to another; this is to represent how jobs in actual real-time systems stay on the same processor during their runtimes.

The queued jobs array has no fixed size, and can be completely empty. Jobs that have been released but are not currently running are stored in this array. The queued jobs array is where most sorting for testing priority occurs.

3 The simulation

We now begin a full overview of the simulator's actual code, and how each piece of the simulator's main loop functions. For the sake of presenting related functionalities of the code together, some parts of the program may be described out of the order in which they appear in the code.

3.1 Initializing the simulator for a task set

Simulating a schedule takes place in a class, `Simulator`, which takes a task set as its only input. `Simulator` uses the task set to establish some useful constants for the eventual simulation:

```
class Simulator:
    def __init__(self, task_set):
        self.task_set = task_set
        self.task_set.sort(key=lambda i: i.id)
        self.release_times = [(t.period, t.phase) for t in self.
    task_set]
        self.hyperperiod = reduce(lambda a, b: a * b // math.gcd(a,
     b), [t.period for t in self.task_set])
        self.max_time = self.hyperperiod + max([t.phase for t in
    self.task_set]) # time at which task set is determined to be
    schedulable
```

release_times. A list of tuples consisting of each task's ϕ_i and T_i. This is kept separate from the main list of tasks mostly for sorting purposes.

hyperperiod. The task set's hyperperiod. The least common multiple of the task set's periods.

max_time. An integer representing a time instant. If the simulator reaches this instant without a deadline being missed, the the task set will be declared schedulable and the simulation will end. If $\phi_i = 0$ for all i, this is equivalent to the hyperperiod, but if not, it is the hyperperiod plus the task set's maximum phase.

3.2 Initializing the simulation

Note that `Simulator` and "simulation" refer to different things. `Simulator` is a class containing a task set to be simulated, a simulation is what actually simulates the task set according to parameters unique to a particular simulation. For example, this makes it easy to simulate a task set once, and then simulate it again under a different scheduling algorithm or context-switching cost.

```
def simulate(self, algorithm, num_procs=1, context_switch_cost=0,
    prin=0, interval1=False):
        self.time = 0 # initialize time
        self.queued_jobs = [] # array of released, inactive jobs
        self.num_procs = num_procs
        self.active_jobs = [None for i in range(self.num_procs)] #
    currently running jobs
        self.context_switch_cost = context_switch_cost
```

The simulation's most relevant arguments are the scheduling algorithm to be used, the number of processors, and *csc*. Before it begins actually simulating the task set, it initializes the active jobs and queued jobs arrays. Note that the active jobs array is never empty, instead leaving `None` in a position if the processor corresponding to it is not being used. It also initializes `time`, which represents the simulation's current time instant.

3.3 The main loop, and checking the status of current jobs

```
while self.time <= self.max_time:
            self.np_decision = False
```

The main body of the simulation consists of a while loop, running until the the simulation time surpasses `max_time`. Each loop also initializes `np_decision`, a Boolean used when the simulation is using a nonpreemptive scheduling algorithm. This variable returns whether a nonpreemptive task can be scheduled at the current time instant.

In the first major block of code in the main loop, the simulation examines the status of the jobs in the active jobs array. It removes completed jobs from the array, and checks for any missed deadlines among the active jobs.

```
for i in range(self.num_procs):
    if self.active_jobs[i] is not None:
        # remove jobs which have finished executing from active
    jobs array
        if self.active_jobs[i].exec_time == 0:
            self.np_decision = True
            if self.context_switch_cost > 0:
                if type(self.active_jobs[i]) is ContextSwitch:
                    self.active_jobs[i] = self.active_jobs[i].link
                else: self.active_jobs[i] = ContextSwitch(self.
    context_switch_cost, False, self.active_jobs[i].task_id)
            else: self.active_jobs[i] = None
        # check if active job has missed deadline
        elif type(self.active_jobs[i]) is not ContextSwitch and
    self.active_jobs[i].deadline_time <= 0:
            if prin > 1: self.printState()
            if prin > 0: print("Missed deadline at", self.time  +
    self.active_jobs[i].deadline_time)
            return False
    else: # processor is open for new job
        self.np_decision = True
```

The simulation iterates over each processor (the active jobs array). If the processor contains a currently active job $J_{i,n}$, the simulation checks whether $RC_{i,n} = 0$, meaning the job has completed its execution time. If this is the case, the job can be safely removed from the active jobs array, and its position in the array is set to `None` or an outgoing context switch depending on whether context-switching costs exist in the simulation. The simulation then checks for whether $RD_{i,n} <= 0$, meaning that $J_{i,n}$ has missed its deadline. If any job at any point in the main loop is found to have missed its deadline, the simulation will immediately end and report where a deadline was missed.

If the processor contains a context switch CS_j, the simulation checks whether $RC_j = 0$, meaning the context switch has run through *csc*. If this is the case, CS_j is removed from the active jobs array and replaced with the job or context switch, or `None` linked to by L_j.

Otherwise, the simulation sets `np_decision` to `True`. It does this in the prior two cases as well, if any active job or context switch have finished running.

The simulation then checks for whether any of the jobs in the queued jobs array has missed its deadline:

```
    # check if queued job has missed deadline
    for job in self.queued_jobs:
        if job.deadline_time <= 0:
            if prin > 1: self.printState()
            if prin > 0: print("Missed deadline at", self.time +
    job.deadline_time)
            return False
```

Next, the simulation checks for whether the current time instant matches any task's release time. If so, a new job instance of that task or tasks is added to the queued jobs array.

```
    # Check if it is any task's release time. If so, add a job of
    that task to queued_jobs
    for task in self.task_set:
        if (self.time - task.phase) % task.period == 0:
            self.queued_jobs.append(Job(task.deadline, task.cost,
    task.id))
```

3.4 Applying scheduling algorithms

Up until this point (except one initialization case involving PEDF), nothing the simulation does depends on the chosen scheduling algorithm. Now we have reached the part of the code in which the simulation makes scheduling decisions based on its scheduling algorithm.

First, the simulation chooses the appropriate job-sorting criteria (the key for a Python sort operation) for each scheduling algorithm.

```
    # Sets task sorting criteria based on scheduling algorithm.
    key = None
    if algorithm in {"EDF", "GEDF", "NPEDF", "GNPEDF", "PEDF"}: key
     = lambda j: j.deadline_time
    if algorithm == "LLF": key = lambda j: j.deadline_time - j.
    exec_time
    if algorithm == "RM": key = lambda j: self.task_set[j.task_id].
    period
```

The earliest deadline first-based algorithms will sort jobs in ascending order by their remaining time until their deadlines, $RD_{i,n}$. The least laxity first algorithm will sort jobs in ascending order by laxity, $RD_{i,n} - RCi, n$. The rate monotonic algorithm will sort jobs in ascending order by the length of their parent tasks' periods, T_i.

One helper function that each of the scheduling algorithm implementations use is `contextSwitchCheck`, used when the simulation is assigning a job to the active jobs array.

```
    # returns a job, incoming context switch, or outgoing context
    switch for assignment to active_jobs
    def contextSwitchCheck(self, old_job, new_job):
        if self.context_switch_cost == 0: return new_job
        incoming = ContextSwitch(self.context_switch_cost, True,
    new_job.task_id, new_job)
        if old_job is None: return incoming
        return ContextSwitch(self.context_switch_cost, False,
    old_job.task_id, incoming)
```

The purpose of `contextSwitchCheck` is to avoid a lot of reused code surrounding whether a context switch needs to be used, and if so, what kind of context switch. It returns a job or context switch to be assigned to the active jobs array. The function takes as arguments `old_job`, which can be a job being preempted or `None` if a job is being assigned to an empty processor, and `new_job`, the job to be assigned to the processor.

First, the simulation checks whether $csc = 0$. Obviously, if this is the case, then no context switch objects are necessary, and the function simply returns `new_job`. Otherwise, the simulation creates a new incoming context switch linking to `new_job`. If `old_job` is `None`, the new job is not preempting another job, so there is nothing that must first be context-switched out of and the function can output the incoming context switch. If `old_job` is not `None`, the new job is preempting another job, so the function outputs an outgoing context switch linking to the incoming context switch. Here we see the importance of the context switch object; stringing context switches and jobs together in this linked-list architecture enables preemptive context-switching without complex alterations of jobs' deadlines and execution times.

3.5 Preemptive scheduling algorithms

Next, the simulator actually applies the sorting algorithms, with algorithms which allow for similar implementations being grouped together. We begin with the implementation that is essentially the default case, applying for EDF, RM, LLF, and GEDF– all of the preemptive algorithms except PEDF.

```
# combine (non-none/context switch) active jobs and queued jobs
 into sorted array
all_jobs = list(filter(lambda j: j is not None and type(j) is
not ContextSwitch, self.active_jobs)) + self.queued_jobs
all_jobs.sort(key=key)
ncs = self.numContextSwitch()
# queued_jobs = each job except the top (num_procs - number of
context switches) in priority
self.queued_jobs = all_jobs[self.num_procs - ncs:]

intersect = [] # all jobs in both active_jobs and newly sorted
queued_jobs
new_jobs = [] # all newly sorted queued jobs not in intersect
for job in all_jobs[:self.num_procs - ncs]:
    if job in self.active_jobs:
        intersect.append(job)
    else:
        new_jobs.append(job)

if len(new_jobs) > 0:
    for i in range(self.num_procs):
        if self.active_jobs[i] not in intersect and type(self.
active_jobs[i]) is not ContextSwitch:
            # active jobs not in intersect are replaced at
their current processor
            self.active_jobs[i] = self.contextSwitchCheck(self.
active_jobs[i], new_jobs.pop(0))
```

```
                if len(new_jobs) == 0:
                    break
```

These algorithms all work on the principle of sorting both the active jobs and the queued jobs as one unit, and assigning the first n jobs of the sorted result to the active jobs array, with n being the number of processors not currently context-switching. The only difference between the different scheduling algorithms here is the sorting `key` assigned to it earlier– in fact, EDF and GEDF are identical as far as the simulation is concerned.

Specifically, we know exactly which jobs will be sorted into the active jobs array as soon as the list of all jobs is sorted. Because of this, we can set the queued jobs array to consist of all jobs past the nth item in the list of all jobs.

The rest of the code mostly ensures that active jobs which are not being preempted remain in their current processor, as moving directly from processor to processor is not allowed. To accomplish this, the simulation constructs two lists: `intersect`, which consists of all active jobs that are not being preempted, and `new_jobs`, which consists of all previously queued jobs which are set to become active. Iterating over each processor slot represented by `active_jobs` until all new jobs have been assigned, the simulation assigns each new job to the first non-occupied processor it finds.

3.6 Nonpreemptive scheduling algorithms

As discussed previously, if the simulator uses a nonpreemptive scheduling algorithm, it will only assign queued jobs to the active jobs array if at least one processor is not being used (represented by `np_decision`).

```
    elif self.np_decision: # assign jobs only if at least one
    processor is available
        self.queued_jobs.sort(key=key) # always EDF
        if len(self.queued_jobs) > 0:
            for i in range(len(self.active_jobs)):
                if self.active_jobs[i] is None:
                    self.active_jobs[i] = self.contextSwitchCheck(
    self.active_jobs[i], self.queued_jobs.pop(0))
                    if len(self.queued_jobs) == 0:
                        break
```

If this is the case, the simulator sorts the queued jobs array by `key` (which, in practice, is always EDF-sorting). It then iterates over each processor, popping the first job in the queued jobs array and assigning it to the first available processor. The loop ends early if there the queued jobs array ever becomes empty.

3.7 Partitioned EDF

The partitioned earliest deadline first scheduling algorithm is unique in that it requires some initialization work set before the main loop. This is because PEDF must assign each task, not individual job instances, to a specific processor. Specifically, it requires an implementation of the bin-packing problem. This problem is NP-hard in the strong sense [1], and instead of testing every possible bin-packing combination the simulator uses the first-fit-decreasing (FFD) [1] heuristic on bins of capacity 1.

```
    # Bin-packing problem implementation
    if algorithm == "PEDF":
        bins = [0 for proc in range(self.num_procs)]
        self.pedf_assignments = [] # stores PEDF processor
    assignment for each task
        densities = []
        for task in self.task_set:
            self.pedf_assignments.append(-1)
            densities.append((task.cost/min(task.period, task.
    deadline), task.id))
        densities.sort(reverse=True)
        packing_fail = False
        for density in densities: # FFD heuristic
            i = 0
            while i <= len(bins):
                if i < len(bins) and (bins[i] + density[0] <= 1 or
    packing_fail):
                    bins[i] += density[0]
                    self.pedf_assignments[density[1]] = i
                    break
                elif i == len(bins):
                    packing_fail = True
                    bins[0] += density[0]
                    self.pedf_assignments[density[1]] = 0
                i += 1
        if packing_fail and prin > 0: print("FFD could not pack the
     task list!")
```

First, the bin-packing implementation initializes itself. It initializes a list of bins (one for each processor), a list of each task's density $\delta_i = C_i \min(T_i, D - i)$ sorted in descending order, and a list to contain the processor assignment of each task `pedf_assignments`.

This is followed by the actual bin-packing. The FFD heuristic iterates over each task's density in descending order, and assigns it to the first bin (processor) into which it can fit [1]. If the task's density can fit into the bin, it is added to total density of the bin, and FFD moves on to the next task. If the task's density cannot fit into any bin, PEDF automatically assigns the task and all tasks thereafter to processor 0. In this case, it does not actually matter which processor the task is assigned to– a missed deadline will result regardless, and changing the processor assignment can only alter when the missed deadline occurs.

Now we can return to the main loop and discuss how the simulation implements PEDF.

```
    if algorithm == "PEDF":
        # combine (non-none/context switch) active jobs and queued
    jobs into sorted array
        all_jobs = list(filter(lambda j: j is not None and type(j)
    is not ContextSwitch, self.active_jobs)) + self.queued_jobs
        all_jobs.sort(key=key)
        if len(self.queued_jobs) > 0:
            proc_check = [i for i in range(self.num_procs)] # list
    of unassigned processors
            i = 0
            while len(proc_check) > 0 and len(all_jobs) > 0 and i <
     len(all_jobs):
                p = self.pedf_assignments[all_jobs[i].task_id] #
    assigned processor ID of task to be scheduled
                if p in proc_check:
                    if type(self.active_jobs[p]) is ContextSwitch:
    i += 1
                    else: self.active_jobs[p] = self.
    contextSwitchCheck(self.active_jobs[p], all_jobs.pop(i))
                    proc_check.remove(p)
                else: i += 1
            self.queued_jobs = all_jobs
```

Similarly to when using the other preemptive scheduling algorithms, the simulation creates a combined list of all active and queued jobs and sorts it EDF-wise in ascending order. However, unlike the other preemptive algorithms, with PEDF the simulator cannot know in advance that the active jobs array will consist of the first n jobs in the sorted list of all jobs. For example, suppose a simulation is using two processors, P_0 and P_1, and that the sorted list of all jobs consists, in order, of a job $J_{0,n}$ whose parent task is partitioned to P_0, another job $J_{1,n}$ whose parent task is partitioned to P_0, and finally a job $J_{2,n}$ whose parent task is partitioned to P_1. Under PEDF, the simulator assigns $J_{0,n}$ to P_0, skips over $J_{1,n}$ because its partitioned processor is now occupied, and finally assigns $J_{2,n}$ to P_1. The array `proc_check` is a list of the processors which have not been assigned.

The simulation makes these assignments by iterating over each job $J_{i,n}$ in the sorted list of jobs, and assigns the ID of the job's parent task's partitioned processor to `p`, which is equivalent to the element of `active_jobs` that represents the processor. If processor P_p has not had a job assigned to it, the simulation assigns $J_{i,n}$ to P_p. Here, `i` represents the number of jobs that have been "skipped" because their partitioned processor already had a job assigned to it, and `i` increases by 1 if a job is skipped. This enables the simulator to pop the first eligible job it can find from the sorted array of jobs.

3.8 The interval

Once the simulation has populated `active_jobs` and `queued_jobs` based on its current scheduling algorithm, it must now simulate time moving forward and adjust each job's $RD_{i,j}$ and, if active, $RC_{i,j}$ accordingly.

Simulating each individual time unit of the task set's hyperperiod would be far too computationally expensive for practical purposes. Instead, we only simulate $t = 0$ and each time instant in which a job or context switch is released, completed, or has its deadline, which we refer to as *decision points*. The interval is the length of time until the next decision point, during which the simulator will not need to make any new scheduling decisions.

We calculate the interval:

```
    if interval1: interval = 1
    # interval = time until next job release
    else: interval = min([rt[0] - ((self.time - rt[1]) % rt[0]) for
     rt in self.release_times])
    if algorithm == "LLF" and not interval1 and self.active_jobs[0]
     is not None and type(self.active_jobs[0]) is not ContextSwitch
     and len(self.queued_jobs) > 0:
        # interval = min(interval, time until a job's laxity
    becomes lower than the active job's)
        interval = min(interval, max(1, min([job.deadline_time -
    job.exec_time for job in self.queued_jobs]) - (self.active_jobs
    [0].deadline_time - self.active_jobs[0].exec_time)))
```

`interval1` is a debugging argument which guarantees that the interval is 1. Otherwise, the simulation sets the interval by calculating the time until the release of the next job, which at time t is $\min(T_i - ((t - \phi_i) \bmod T_i)$ for all τ_i). Note that this may not be the final interval, the next decision point may be sooner than that.

Using the LLF scheduling algorithm requires a special case. This is because, unlike all the other implemented scheduling algorithms, under LLF individual job's priorities are be dynamic. The lengthy expression used in this case sets the interval to the time until another job's laxity becomes less than that of the current active job, if that time is less than the previously calculated interval. (This case refers to `active_jobs[0]` because the simulator does not implement multiprocessor LLF, so there can only ever be one active job.

The last step of calculating the interval is to set it to the smaller of the previously calculated interval and the remaining time until any job or context switch either finishes executing or reaches its deadline.

```
    active_non_none = list(filter(lambda j: j is not None, self.
    active_jobs))
    if len(active_non_none) > 0:
        # interval = min(interval, time until a job/context switch
    finishes executing or reaches its deadline)
        interval = min(min([min(j.exec_time, j.deadline_time) if
    type(j) is Job else j.exec_time for j in active_non_none]),
    interval)
        for job in self.active_jobs:
            if job is not None:
                job.exec_time -= interval
                if type(job) is not ContextSwitch: job.
    deadline_time -= interval
```

Once the final interval is calculated, the above block of code iterates through the active jobs array and subtracts the interval from each active job $J_{i,n}$'s remaining deadline time $RD_{i,n}$ and remaining execution time $RC_{i,n}$, and each context switch CS_j's remaining execution time RC_j.

Finally, the simulation iterates through the queued jobs array and subtracts the interval from each queued job $J_{i,n}$'s remaining deadline time $RD_{i,n}$:

```
    for job in self.queued_jobs:
        job.deadline_time -= interval
    interval = max(1, interval)
    self.time += interval
    if prin > 2: input() # pause between decision points
```

It then adds the interval to the current time, making the new current time equal to the next decision point. Finally, the main loop returns to the start, and makes its next scheduling decisions at the new time instant.

4 User Experience

The simulator runs as a Python script using the built-in Python `input()` function. We will go over an example that details how to use the schedule simulator.

4.1 Constructing a task set

The simulator's interface begins by asking the user how they would like to to construct a task set:

```
Welcome to Eli Zachary's scheduling simulator.
Would you like to [m]anually enter a task set, [l]oad a task
set from task_set.json, [r]andomly generate a single task set,
or randomly generate multiple task [s]ets until a schedulable
one is found?
$
```

Suppose the user chooses to manually enter a task set. In this case, the user specifies $\boldsymbol{\tau}_i = (\phi_i, T_i, C_i, D_i)$. (The task ID is automatically generated and cannot be chosen by the user.)

```
$ m
Task 0 phase: 0
Task 0 period: 10
Task 0 cost: 5
Task 0 deadline: 10
[T 0, PHASE 0, PERIOD 10, COST 5, DL 10]
Would you like to [a]dd another task, [r]edo the current task,
or [c]omplete the task set?
$
```

Once a task has been created, the script will display the task before given the user a chance to redo its entry if any mistakes were made. The user can add as many tasks as they want.

```
Would you like to [a]dd another task, [r]edo the current task,
or [c]omplete the task set?
$ a
Task 1 phase: 0
Task 1 period: 6
Task 1 cost: 10
Cost must be less than or equal to the task's period of 6
Task 1 cost:
$
```

If the user has entered a value that would make the task set blatantly impossible to schedule, such as having an execution cost larger than the task's period, the simulator will detect it and prompt the user to enter an acceptable value.

```
Would you like to [a]dd another task, [r]edo the current task,
or [c]omplete the task set?
$ a
Task 1 phase: 0
Task 1 period: 6
Task 1 cost: 10
Cost must be less than or equal to the task's period of 6
Task 1 cost:
$
```

Here, the user enters an acceptable value and then chooses to complete the task set. Once the task set is completed, the simulator will display the task set. Now the simulator can create its instance of the actual `Simulator` class.

```
Cost must be less than or equal to the task's period of 6
Task 1 cost: 3
Task 1 deadline: 6
[T 1, PHASE 0, PERIOD 6, COST 3, DL 6]
Would you like to [a]dd another task, [r]edo the current task,
or [c]omplete the task set?
$ c
[[T 0, PHASE 0, PERIOD 10, COST 5, DL 10], [T 1, PHASE 0,
PERIOD 6, COST 3, DL 6]]
```

Entering large task sets this way, especially if one wants to make minor adjustments, can get very tedious. The user can choose to load a task set in JSON format from `task_set.json`, which can be edited much more conveniently. Alternatively, the user can generate a random task set consisting of a chosen number of randomly generated tasks. These tasks are generated with values randomly chosen from presets for each of (ϕ_i, T_i, C_i, D_i). The task sets constructed this way tend to have overly long hyperperiods.

4.2 Simulation parameters

Once the user has created a task set, the task set's schedulability can be tested in a variety of different ways. The user can set parameters including the scheduling algorithm, the context-switching cost, and how much information about the simulation they'd like to have printed.

Here, the user preloads a task set, and then decides to test it using EDF, with a context-switching cost of 0, and printing the output at every decision point.

```
Would you like to [m]anually enter a task set, [l]oad a task
set from task_set.json, [r]andomly generate a single task set,
or randomly generate multiple task [s]ets until a schedulable
one is found?
$ l
[[T 0, PHASE 0, PERIOD 10, COST 5, DL 10], [T 1, PHASE 0,
PERIOD 15, COST 4, DL 20], [T 2, PHASE 0, PERIOD 30, COST 10,
DL 30]]
```

```
    Which scheduling algorithm would you like to use?
    Options include: {'LLF', 'PEDF', 'RM', 'GEDF', 'NPEDF', 'GNPEDF
    ', 'EDF'}
    $ edf
    Context-switching cost: 0
    Select print mode.
    0: Print nothing at all. 1: Print whether the task set is
    schedulable under the chosen algorithm. 2: Print the schedule
    at all decision points.
    3. Pause simulation at each decision point..
    $ 2
```

The simulator will then print the result of simulating the task set, with the simulation's status at each decision point printed out. I am not going to include any additional snapshots of console results in the interest of keeping this paper a reasonable length, but the user can them re-simulate the task set with any changes to the simulation parameters.

Instead of simulating a single task set, the user can also choose to simulate an arbitrary amount of randomly generated task sets over some given simulation parameters, with the simulations ending once either a task set schedulable according to the parameters has been generated. The simulator will then either print out the one schedulable task set and report how many iterations it took to generate it, or print that it was unable to find a schedulable task set within the arbitrary amount of task sets. This is useful for testing the limits of various simulation parameters– for instance, LLF and context-switching costs above 0 rarely mix well together.

5 Limitations and conclusion

There is a great deal of room for improvement in my schedule simulator. A useful improvement would be compatibility with floating point task parameters instead of requiring all of them to be integers, for example. I definitely cannot claim that these are the most efficient implementations of the scheduling algorithms, and I'm sure my implementation would not hold up compared to other solutions in a real-world setting.

I had hoped to be able to implement PropShare and PFair algorithms, but both of those would require very different implementations from the scheduling algorithms I did implement, and ultimately they would have extended the paper to be too long to be worth it. As the project description suggested, creating graphical representations of schedules would have been an exciting feature to add, though I knew from the beginning that it would very likely not be worth the amount of time required to implement.

Still, the scheduling simulator as it exists is a useful tool. It has a great deal of functionality, and allows the user to observe in detail the circumstances under which a task set is schedulable or not schedulable. When one implements a scheduling algorithm, they must have a full understanding of how it works, not just in theory, but in practical situations. So if nothing else, I have created a solid educational tool, and that in itself was a worthwhile cause.

References

[1] Sudarshan K. Dhall and C. L. Liu. “On a Real-Time Scheduling Problem”. In: *Operations Research* 26.1 (1978), pp. 127–140.

[2] J.W.S. Liu. *Real-Time Systems*. Prentice Hall, 2000. ISBN: 9780130996510.