Istanbul Technical University

Computer Engineering Department

BLG 450E

Real Time Systems Software

HOMEWORK REPORT

(Simulating Real Time Scheduling Algorithms)

Author:
Furkan Kalay
150220049

Date:
December 1, 2025

Fall 2025-2026

1 Introduction

In this homework, the objective is to simulate the real-time scheduling algorithms taught in the course. These algorithms include Rate Monotonic (RM), Deadline Monotonic (DM), Earliest Deadline First (EDF), and Least Laxity First (LLF) for periodic tasks; and Background, Polling, and Deferrable Servers for aperiodic tasks. There were no programming language restrictions, so the implementation follows C++ standards.

2 Assumptions

In this homework, several assumptions are made. First of all, all time intervals are taken as 1, which are atomic. It means that all timing parameters(like release time, period, deadline, etc.) are integers. Simulation will not work for floating numbers.
Other than that, it is assumed that the user will run the program in a terminal with the following format: exe file-input file-algorithm for only periodic tasks, exe file-input file-algorithm-server-budget-period for aperiodic tasks (budget and period can be ignored for background server). Also, the input file only parsed for the following format as described in the homework description:
Comment line
P ri ei pi di
P ri ei pi
P ei pi
D ei pi di
A ri ei (Background, Poller, Defereable Server)
They will be handled as ri=0 if not given and di=pi if also not given. Finally, all the tasks and servers are made up into the following struct with some arrangements:

```
struct Task{
    string id;
    int release_time;
    int exec_time;
```

```
    int period;
    int deadline_relative;
    int absolute_deadline;
    float priority=0.0;
    bool isActive=false;
    int remaining_exec = 0;
    Task() = default;
    Task(string id_, int r, int e, int p, int d_rel)
        : id(id_), release_time(r), exec_time(e),
          period(p), deadline_relative(d_rel)
    {
        absolute_deadline = release_time + deadline_relative;
        remaining_exec    = exec_time;
    }
    Task(string id,int r,int e):id(id),release_time(r),exec_time(e){}

};
```

The final assumption made is that it is sufficient if we can see algorithm can run until the hyperperiod. It is not necessary to run the simulation until aperiodic tasks are completed. If the server is running all the time successfully till the hyperperiod, it means it will eventually complete its aperiodic tasks.

3 Periodic Algorithms

3.1 Rate Monotonic

The first algorithm implemented is rate monotonic. This part is important as all the foundations for other periodic tasks will be discussed here. Since this part includes all periodic tasks' behavior, the half of the basis for aperiodic tasks will also be discussed here.

```
void rateMonotonic(vector<Task> &tasks){
    if(!checkFeasibility(tasks)){
        cout<<"This task set is not schedulable";
        return;
    }
     for (auto &t : tasks) {
        t.priority       = 1.0f / t.period;
        t.absolute_deadline = t.release_time + t.deadline_relative;
        t.remaining_exec = t.exec_time;
```

```
        t.isActive       = false;
    }

    int hyperperiod=calculate_hyperperiod(tasks);
    int firstSimoultaneous=findFirstSimultaneousRelease(tasks);
    vector<Task> currentTasks=tasks;
    for(int time=0; time<hyperperiod+firstSimoultaneous; time++){
        for(auto &ct:currentTasks){
            if(ct.absolute_deadline<=time){
                cout<<"Deadline missed for "<< ct.id<<" at " <<time <<".
    Algorithm is failed.\n";
                return;
            }
        }

        for (auto &task : currentTasks) if(task.release_time<=time)task.
    isActive=true;

        sortByExplicitPriority(currentTasks);
        int runningTaskIndex=0;

        for (auto &task : currentTasks){
            if(task.isActive==false)runningTaskIndex++;
            else break;
        }
        if(runningTaskIndex==currentTasks.size()){
            cout<<time<<" is IDLE\n";
            continue;
        }
        cout<<"At time: "<< time<<" Task" << currentTasks[
    runningTaskIndex].id<<" is running\n";

        currentTasks[runningTaskIndex].remaining_exec--;
        if(currentTasks[runningTaskIndex].remaining_exec==0){
            Task oldTask=currentTasks[runningTaskIndex];
            Task newTask=oldTask;

            newTask.isActive=false;
            newTask.release_time=oldTask.release_time+oldTask.period;
            newTask.absolute_deadline=newTask.release_time+newTask.
    deadline_relative;
            newTask.remaining_exec=newTask.exec_time;

            currentTasks.erase(currentTasks.begin()+runningTaskIndex);
            currentTasks.push_back(newTask);
        }
```

```
52      }
53      cout<<"Successfully scheduled until hyperperiod.\n";
54  }
```

After getting a periodic tasks array as a parameter, the first thing I do is feasibility checking. If CPU utilization is higher than 1, then there is no point in seeing scheduling, as this will violate hard real-time system rules. After that parameters for tasks are managed. isActive flag is false if the task in the array has not been released, and it will be checked continuously in the for loop. Then, hyperperiod and first simultaneous time are calculated via helper functions because I must see tasks from their first simultaneous initialization to the next initialization to see if they can be scheduled or not. Next, simulating the schedule started in the for loop. I check if a deadline is missed for any task first. Then, update isActive flag to see if they are released yet or not. The following part is critical as I sort the task array according to their priorities, which is 1/period, with a helper function. With the following for loop, I ensure that the highest priority active task will be executed. Also, it is clear that if I cannot find a proper index for current tasks, it means that the time slice is idle. I am simulating the execution of the task by decreasing remaining_exec parameter. And, I should check if that parameter is 0 or not. If so, it means that I should delete that instance and add a new one with proper parameters. This loop will handle rate monotonic scheduling eventually.

3.2 Deadline Monotonic

As DM and RM are both static priority scheduling algorithms their scheduling is simple, I just change the priority calculation with 1/relative deadline.

3.3 Earliest Deadline First

Even if RM and DM are both different from EDF in terms of priority calculation, as EDF is a dynamic priority. But our implementation for sorting the array makes everything almost identical. Since I are adding instances, not tasks themselves, to the array at different times, I have to

sort it every time when a new instance is added (it is literally every time in our implementation). In short, only difference between RM, DM and EDF is their priority calculations. I sort the array in each time interval with ascending absolute deadline order.

3.4 Least Laxity First

As mentioned before, all descriptions are identical to EDF. Even if LLF is more dynamic than EDF (LLF is dynamic in terms of both tasks and instances, EDF is static in terms of instances), thanks to sorting in each for loop, I just changed priority calculation again.
task.priority = task.absolute_deadline - task.remaining_exec - time;

4 Aperiodic Algorithms

4.1 Background Server

Background server is the simplest server algorithm. It just checks idle times to run arrived aperiodic tasks. I get and handled aperiodic tasks with a seperated array.

```
    for(auto &at: aperiodicTasks){
        at.absolute_deadline=INT_MAX;
        at.period=INT_MAX;
        at.deadline_relative=INT_MAX;
        at.isActive=false;
        at.remaining_exec=at.exec_time;
    }
```

With the following code, I adapted aperiodic tasks to a regular task struct as they have no deadline or recurrence. I made these parameters maximum.

```
        if(choose==1 || choose==2) sortByExplicitPriority(currentTasks);
        if(choose==3)sortByEarliestDeadline(currentTasks); // 3 and 4
   are dynamic priorities
        if(choose==4){
            for (auto& task : currentTasks) task.priority = task.
   absolute_deadline - task.remaining_exec - time;
            sortByLeastLaxity(currentTasks);
        }
```

With those codes I implemented 4 different algorithms to the server with choose parameters provided by the user.

```
        if(runningTaskIndex==currentTasks.size()){
            cout<<"System can run aperiodic task at time:"<< time <<".\n
    ";
            if(currentAperiodics.empty()){
                cout<<"But there is no aperiodic Task. It is IDLE time\n
    ";
                continue;
            }
            sortTasksByReleaseTime(currentAperiodics); // first come
    first serve(the one first came at the first index of array)
            if(currentAperiodics[0].release_time<=time){
                currentAperiodics[0].remaining_exec--;
                cout<<"Aperiodic Task: "<<currentAperiodics[0].id<<" is
    running.\n";
                if(currentAperiodics[0].remaining_exec==0){
                    currentAperiodics.erase(currentAperiodics.begin());
                }
            }else{
                cout<<"But no aperiodic task released yet. It is IDLE
    time\n";
            }
            continue;
        }
```

As I mentioned before, we know when we hit to idle time. At idle time we can finally execute our aperiodic tasks. As it is visible from the code above we apply same rules except appending new instance to array. The rest of the algorithm is the same with other periodics. Only idle time hit handled and I get the simplest server algorithm.

4.2 Polling

Codes are getting more complex in here. First of all I added new Server struct to control server information.

```
struct Server{
    string ID;
    int period;
    int budget;
    Server(string id,int p,int b):ID(id),period(p),budget(b){}
};
```

Then I adapted it to a regular task by adjusting its execution time with budget and deadlines with period. Then I pushed into the tasks array.

```
        for(auto &ct:currentTasks){
            if(ct.id==poller.ID)continue;
            if(ct.absolute_deadline<=time){
                cout<<"Deadline missed for "<< ct.id<<" at " <<time <<". Algorithm is failed.\n";
                return;
            }
        }
        for (auto &task : currentTasks) if(task.release_time<=time)task.isActive=true;

        if(choose==1 || choose==2) sortByExplicitPriority(currentTasks);
        if(choose==3)sortByEarliestDeadline(currentTasks); // 3 and 4 are dynamic priorities
        if(choose==4){
            for (auto& task : currentTasks) task.priority = task.absolute_deadline - task.remaining_exec - time;
            sortByLeastLaxity(currentTasks);
        }

        if(time%poller.period==0){
            usedBudget=0;
            for(auto & at:currentAperiodics){
                if(at.release_time<=time){
                    if(at.remaining_exec<=poller.budget-usedBudget){
                        usedBudget+=at.remaining_exec;
                    }else{
                        usedBudget=poller.budget;
                    }
                }
                if(usedBudget==poller.budget)break;
            }
            for(auto& t:currentTasks){
                if(t.id==poller.ID)t.remaining_exec=usedBudget;
            }
        }

        int runningTaskIndex=0;
        for (auto &task : currentTasks){
```

```
36              if(task.isActive==false)runningTaskIndex++;
37              else{
38                  if(task.id==poller.ID && task.remaining_exec==0){
39                      runningTaskIndex++;
40                  }else break;
41              }
42          }
43
44          if(runningTaskIndex==currentTasks.size()){
45              cout<<"System is Idle at "<< time <<".\n";
46              continue;
47          }
48
49          Task& running=currentTasks[runningTaskIndex];
50          if(running.id==poller.ID){
51              cout << "At time: " << time << " Server " << running.id
52              << " serving " << currentAperiodics[0].id
53              << " (budget: " << running.remaining_exec << ")\n";
54
55              running.remaining_exec--;
56              currentAperiodics[0].remaining_exec--; //if we have no ready
     aperiodic task server.remaining_exec will be 0 so runningtask index
    will never be here
57              if(currentAperiodics[0].remaining_exec==0){
58                  currentAperiodics.erase(currentAperiodics.begin());
59              }
60              continue;
61          }
```

This 60 lines of code placed beginning of regular for loop which manage simulation. As seen in line 2 we don't care the deadline of server. Following 15 lines are the same. We are handling polling server at line 17. It checks whether there are waiting aperiodic tasks at the beginning of period. If so we consume budget until it finishes. Then we will search for next running task with the highest priority (remember server is also kind of task) if we hit server and some budget reserved (if remaining execution time is 0 it means budget never reserved or consumed) we will execute aperiodic task. Rest of the steps are the same with background server. And following of the code is the same with all other, about executing periodic task and appending new one if necessary.

4.3 Deferrable Server

It is very similar to polling server. Only difference is that we don't check if there are any aperiodic task at the beginning of each period.

```
        if(time%df.period==0){
            for(auto& t:currentTasks){
                if(t.id==df.ID){
                    t.remaining_exec=df.budget;
                    t.release_time=time;
                    t.absolute_deadline=time+df.period;
                    break;
                }
            }
        }
```

In each period time of the server we refresh budget by setting parameters like that. We mimic remaining execution time of task struct as budget.

```
        bool isThereAperiodicTask=false;
        for(auto& task:currentAperiodics){
            if(task.release_time<=time)isThereAperiodicTask=true;
        }

        int runningTaskIndex=0;
        for (auto &task : currentTasks){
            if(task.isActive==false)runningTaskIndex++;
            else{
                if(task.id==df.ID && (task.remaining_exec==0 || !
  isThereAperiodicTask)){
                    runningTaskIndex++;
                }else break;
            }
        }

        if(runningTaskIndex==currentTasks.size()){
            cout<<"System is Idle at "<< time <<".\n";
            continue;
        }

        Task& running=currentTasks[runningTaskIndex];
        if(running.id==df.ID){
            cout << "At time: " << time << " Server " << running.id
            << " serving " << currentAperiodics[0].id
            << " (budget: " << running.remaining_exec << ")\n";

            running.remaining_exec--;
```

```
28              currentAperiodics[0].remaining_exec--; //if we have no ready
    aperiodic task server.remaining_exec will be 0 so runningtask index
   will never be here
29              if(currentAperiodics[0].remaining_exec==0){
30                  currentAperiodics.erase(currentAperiodics.begin());
31              }
32              continue;
33          }
```

We check every time if there are any aperiodic tasks. If there are any we also check if there are remaining budget. If we have budget and aperiodic task we simply execute it like before, if not we continue with regular way of executing periodic tasks.

5 Results

The implemented program tested for many times. Following program executed for P 2 6, P 1 4, P 3 12, P 1 12 input and rate monotonic algorithm.

```
inputs.txt file read succesfully. Periodic Task Count: 4Aperiodic Task Count: 0
Running Periodic Scheduling: rm
At time: 0 TaskT2 is running
At time: 1 TaskT1 is running
At time: 2 TaskT1 is running
At time: 3 TaskT3 is running
At time: 4 TaskT2 is running
At time: 5 TaskT3 is running
At time: 6 TaskT1 is running
At time: 7 TaskT1 is running
At time: 8 TaskT2 is running
At time: 9 TaskT3 is running
At time: 10 TaskT4 is running
11 is IDLE
At time: 12 TaskT2 is running
Successfully scheduled until hyperperiod.
```

Figure 1: Rate Monotonic Example

An example for aperiodic tasks for the following inputs P 2 6, P 1 4, A 1 2, A 4 2

```
PS C:\Users\Furkan\desktop> ./rts.exe inputs.txt edf deferrable 1 3
inputs.txt file read succesfully. Periodic Task Count: 2Aperiodic Task Count: 2
At time: 0 TaskT2 is running
At time: 1 Server ServerTask serving A1 (budget: 1)
At time: 2 TaskT1 is running
At time: 3 Server ServerTask serving A1 (budget: 1)
At time: 4 TaskT1 is running
At time: 5 TaskT2 is running
At time: 6 Server ServerTask serving A2 (budget: 1)
At time: 7 TaskT1 is running
At time: 8 TaskT1 is running
At time: 9 Server ServerTask serving A2 (budget: 1)
At time: 10 TaskT2 is running
System is Idle at 11.
At time: 12 TaskT2 is running
Successfully scheduled until hyperperiod.
```

Figure 2: Deferrable Server / EDF example

6 GUI with Python

With some quick scripts, we visualize the scheduler in Python. Running this command is as easy as possible. With the names of files on my computer I can run it like that: Python visualize_schedule.py rts.exe inputs.txt dm polling 1 4. But it supports only certain inputs. We can run it only by Pyton (python file name) (executable file of my C++ code) (input file name) (periodic scheduler name) (aperiodic server name) (budget) (period) for background only server name is sufficient. An example GUI:

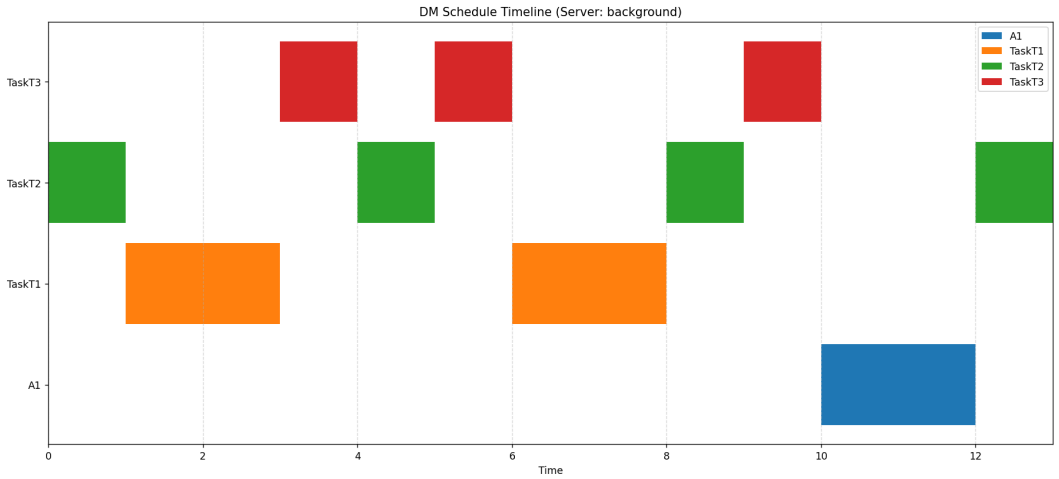

Figure 3: Matplotlib Interface with Background Server

7 Conclusion

In this homework, a comprehensive real-time scheduling simulator was developed using C++. The project covered the implementation of classic periodic algorithms (RM, DM, EDF, LLF) and aperiodic server mechanisms (Background, Polling, Deferrable). Key takeaways from this implementation include:

- **Static vs. Dynamic Priorities:** Implementing RM and EDF highlighted the trade-off between implementation simplicity and scheduling capability. While RM is easier to implement with fixed priorities, EDF allows for 100% CPU utilization, though it requires sorting the ready queue at every scheduling point.
- **Aperiodic Handling:** The simulation confirmed that treating aperiodic servers as periodic tasks with specific capacities is an effective way to integrate them into standard scheduling policies. The Deferrable Server proved to be superior in responsiveness compared to Polling and Background approaches.
- **Simulation Challenges:** Handling atomic time units and ensuring correct state transitions (Active/Idle) required careful management of the simulation loop. The use of Object-Oriented structures in C++ greatly facilitated the management of task states and server budgets.

Overall, the simulator successfully meets the homework requirements and provides a practical insight into how real-time operating systems manage CPU resources under strict timing constraints.

References

[1] Deniz Turgay Altılar, *BLG 450E Real-Time Systems Software Lecture Slides*, Istanbul Technical University, Computer Engineering Department, Fall 2025.