IN4343 Real-Time System Group 24 - Lab1

Yuhang Tian
5219728

Mingyu Gao
5216281

March 5, 2021

1 Part1

1.1 Question1

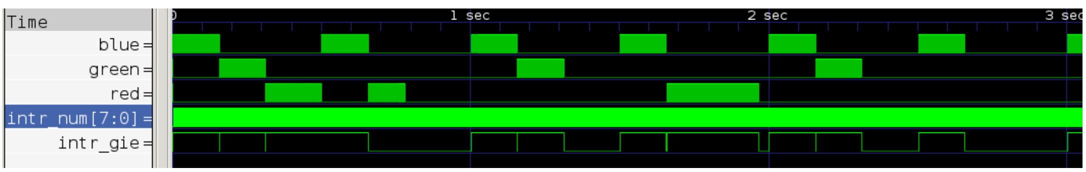
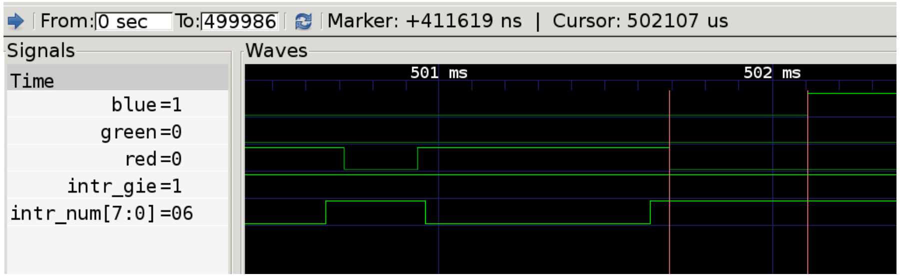


Figure 1: simulation from 0 to 3

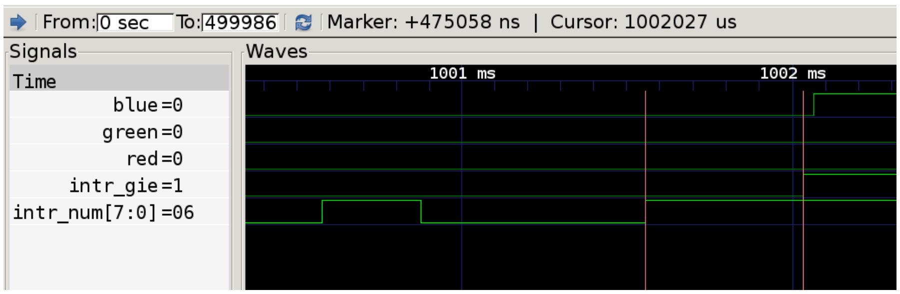
According to the result of stimulation, this pattern is repeated every 3 seconds. Therefore, we only consider the first 3-second block shown in figure 1. In addition, we divide the program block into three intervals, lasting 1 second, and record the execution time for different corresponding functions.

Table 1: System functionality

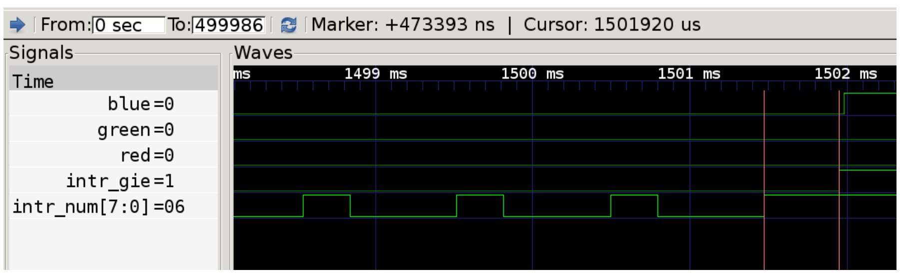
Functionality	Interval	Time (μs)	Description	Hyperperiod Sum (μs)
Context Switch Function	0 to 1	151.791	A switches to B	
	0 to 1	149.081	B switches to C	
	0 to 1	413.619	C switches to A & interrupt	
	0 to 1	195.789	A switches to C	
	1 to 2	151.569	A switches to B	
	1 to 2	212.229	A switches to C	
	2 to 3	150.573	A switches to B	1424.651
Scheduling Function	0 to 1	33.333	scheduling A (measured form the second hyperperiod)	
	0 to 1	32.220	scheduling B	
	0 to 1	338.707	scheduling C&A	
	1 to 2	33.187	scheduling A	
	1 to 2	32.898	scheduling B	
	1 to 2	33.332	scheduling A	
	1 to 2	33.058	scheduling C	
	2 to 3	32.991	scheduling A	
	2 to 3	33.202	scheduling B	
	2 to 3	33.498	scheduling A	636.426
Timer Interrupt Handler	500ms	?	measured beforehand	
	1000ms	475.058		
	1500ms	473.393		
	2000ms	477.062		
	2500ms	434.402		1860.473



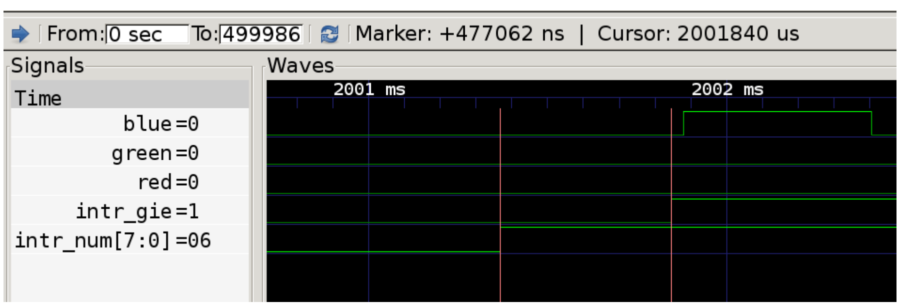
(a) TimeInterrupt500ms



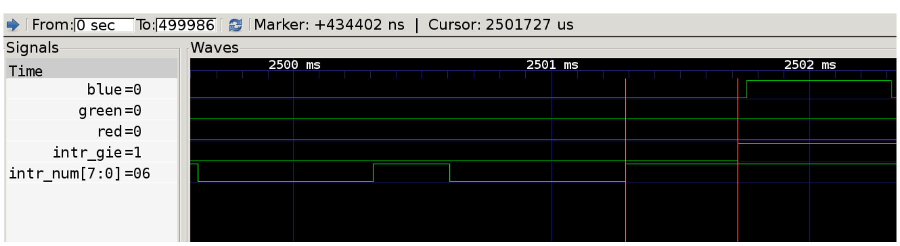
(b) TimeInterrupt1000ms



(c) TimeInterrupt1500ms



(d) TimeInterrupt2000ms



(e) TimeInterrupt2500ms

Figure 2: Hyperperiod TimeInterrupt

1.2 Question2

$$T_{avgBlue} = 155.226ms,\ T_{avgGreen} = 155.223ms,\ T_{avgRed} = 310.393ms$$
$$T_{totalTasks} = 6T_{avgBlue} + 3T_{avgGreen} + 2T_{avgRed} = 2017.811ms$$
$$T_{idle} = 970.6ms$$
$$T_{overhead} = T - T_{totalTasks} - T_{idle} = 11.589ms$$
$$Avg_{overhead} = 43.189ms/3s = 0.386\%$$

Here, the overhead time does not include system idle time.

1.3 Question3

#Task	Blue Time	Green Time	Red Time	Task Time	Idle Time	Overhead Time	Average Overhead(%)
1	136.873	0	0	821.238	2167.626	11.136	0.371
2	145.856	145.815	0	1312.581	1676.127	11.292	0.376
3	155.226	155.223	310.393	2017.811	970.6	11.589	0.386
4	158.348	158.598	316.580	2059.042	929.527	11.431	0.381
5	165.958	165.835	331.630	2156.513	832	11.478	0.383
6	172.004	172.259	343.861	2232.475	755	12.525	0.418
7	178.496	178.005	356.434	2317.859	670.8	11.341	0.378
8	181.633	181.562	363.176	2360.836	625.822	13.342	0.445
9	191.680	191.591	383.664	2492.181	497.667	10.152	0.338
10	200.213	200.161	399.6	2593.2	394.312	12.488	0.416

Table 2: the average overhead vs. the number of Tasks

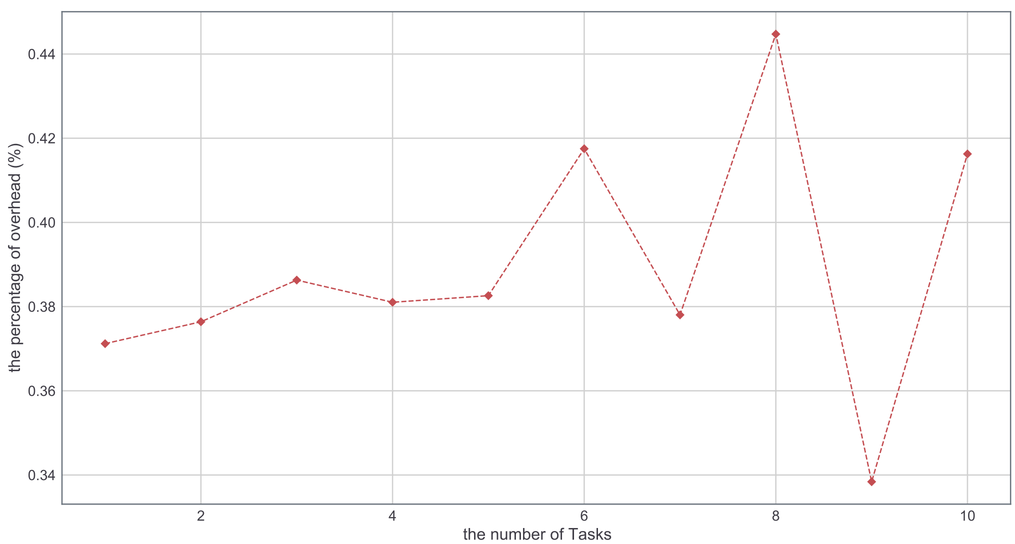


Figure 3: the average overhead vs. the number of Tasks

The average system overhead fluctuates around 0.4% which is almost a constant since it always needs to check if a new job is released at every timer tick no matter how many tasks there are.

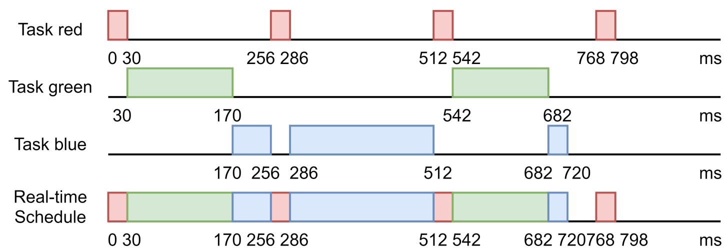
2 Part2

2.1 Question1

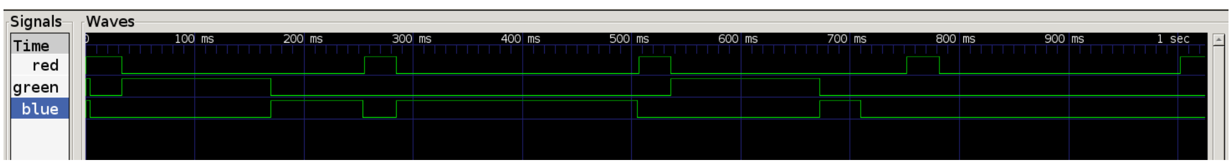
More details are shown in the code.
You can download the code to see the completed version (recommended) or the code in the appendix.

2.2 Question2

We schedule Tst1 by an event-based algorithm and the results are shown in figure 4. Figure 4(a) is the hand-sketch version and figure 4(b) is the simulated one. The results reveal that the simulation is consistent with the theoretical scheduling outcome.



(a) Sketch for Tst1



(b) Scheduling Snapshot for Tst1

Figure 4: Scheduling for Tst1 (Inverted)

Parameter	Description	Initial Value
minPeriod	Minimal period of all tasks.	0xffff
minAt	Remark which task has the minimal period.	null
delta	Spare time for this slot. Initial delta is minPeriod/4 because Period is multiplied by 4 in RegisterTask.	minPeriod / 4
i	Number of tasks.	number of tasks

Table 3: Parameters used in code

Code Description

- **Timer Interrupt Handler**
  Our timer interrupt handler is implemented in **SchedulerOnline.c**, realized by setting the next interruption time (*NextInterruptTime*) as the next closest job-releasing time. Since it is periodic, the closest releasing time is decided by the task with the shortest period.
  - the inserted code is to interrupt and monitor whether there is a task arriving, so the monitoring period is set to the minimum period among the tasks which is the period of task red;
  - the arriving task will be pushed into its queuing line - there is not a real buffer, but a counter to count the pending number and an accumulative timer to record the remaining execution time;
  - the next releasing time for the task which occurs at this interruption will be updated. As the tasks are periodic, the next releasing time can be predicted.

  A loop in *RegisterTask* is set to traverse and obtain the position - *Prio* - of the task with the minimum period.
- **Scheduler**
  As shown in figure 5, the flowchart of the scheduler, a two-layer judgement among all tasks in one whole interrupted time could realize the scheduler with implementation in function *Scherduler_P_FP* in file **Scherduler_P_FP.c**.
  - Start of program: This function uses the number of tasks(i) as input for the next step.
  - Tasks' traversal: The program goes from the last task in the task list because in this real-time scheduler the last has the highest priority.

- Pending jobs check: The program introduces parameters *t->Activated* and *t->Invoked* to check whether this task has pending jobs. Their equality represents no pending jobs for this task and then the program could continue to check the next task with a comparably lower priority.
- Executing unfinished task: This logic is applied in the case that one task has pending jobs, aiming at calculating how much of one unfinished task can be executed in this free time. It first compares remaining time of this task(*t->Remaining*) with spare time(*delta*) left in this slot. When spare time is larger than remaining time, pending jobs in this task can all be executed and remained time can be set back to zero. By parity of reasoning, if remaining time exceeds the spare time, this task can not be completed within this slot, so, this task's remaining time should be updated to *remaining time - delta*. To sum up, both of remaining time(*t->Remaining*) and spare time(*delta*) will be updated in this code block.
- End of program: The scheduler ends when it traverses all tasks and executes them until no more pending job exists during this slot.
- **Missing Deadline**: Since it is fixed priority scheduling, the task which misses its deadline will be executed later if there is spare time.

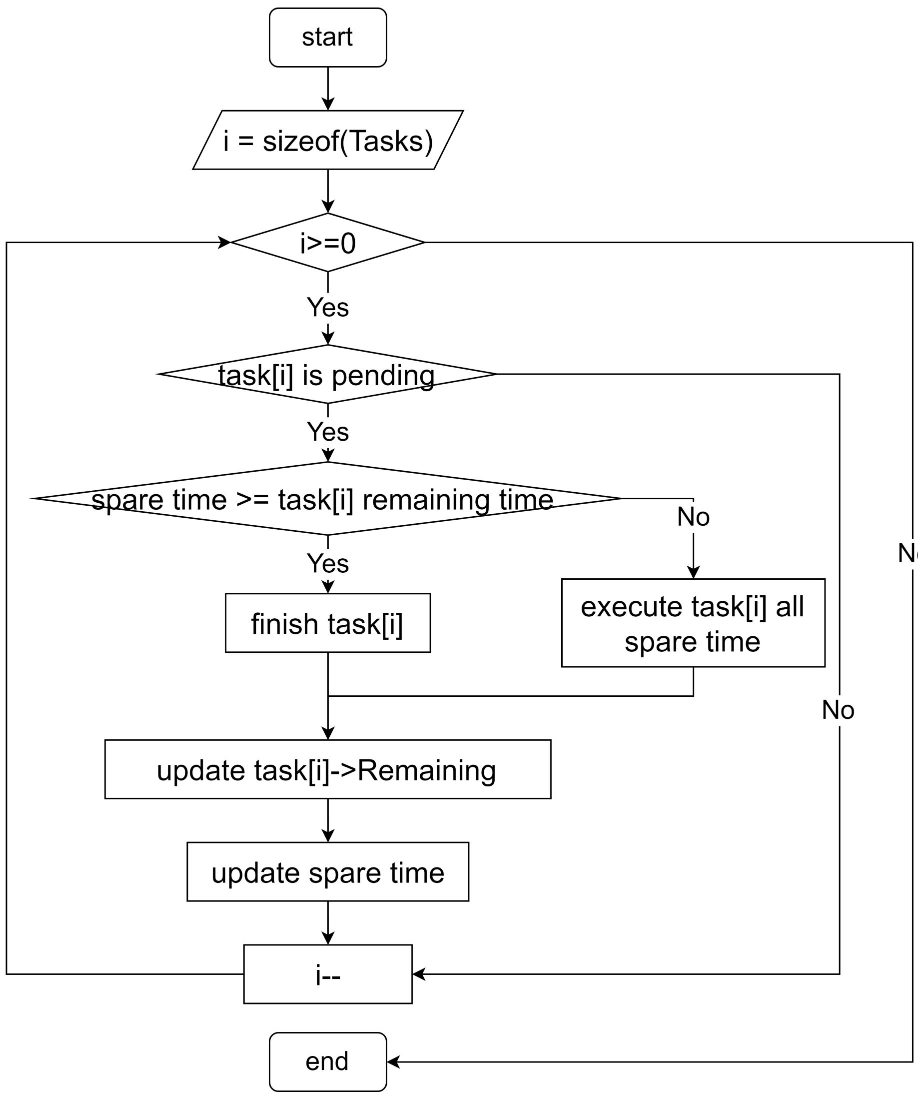


Figure 5: The Flowchart of Scheduler

2.3 Question3

As shown in figure 4 and table 6, the total overhead in one hyperperiod is 2 ms, consisted of 1.5 ms for the interruption and 0.5 ms for scheduling.

The source clock of the counter is Auxiliary Clock (ACLK) which frequency is 32KHz while the divider is set to 8, consequently, the actual frequency of TAR is 4KHz. Therefore, for every 0.25ms, the accumulative 16-bit counter will count 1. Since an output port is merely 8-bit, in order to visualize the result of the timer, it uses two ports to represent higher 8 bits and lower 8 bits separately. In figure 6, port 1 and port 2 reveal the lower and higher bits of the time spending on the interruptions, whereas port 3 and port 4 show the lower and higher bits of the time spending on the scheduling. *It should be noticed that we include ContextSwitch and ContextResume when measuring the time of the interruption.*

Overhead	Count (0.25ms/trigger)	Time Consumption (ms)
Time Interrupt	6	1.5
Scheduler	2	0.5
Total Overhead	8	2

Table 4: Overhead in one Hyperperiod

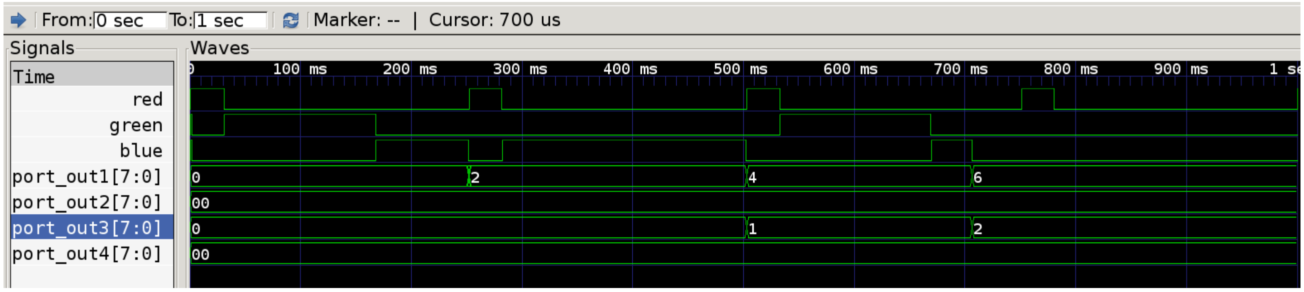


Figure 6: Overhead in one Hyperperiod(Task Inverted)

2.4 Question4

The TimeTracking will interfere with the program but not significantly. It will take up the resources to execute the counting process which is originally used to execute the tasks and their relevant processes such as scheduling. In addition, since the results need to be mapped to the corresponding ports and printed out, it will also take some time to process. However, as their processing speed is fast, in other words, their time consumption is relatively less compared with the time for processing tasks, the interference is not notable.

2.5 Question5

Figure 7 shows how the percentage of overhead in one hyperperiod changes with the number of tasks. It can be seen that the overhead increases with the number of tasks.

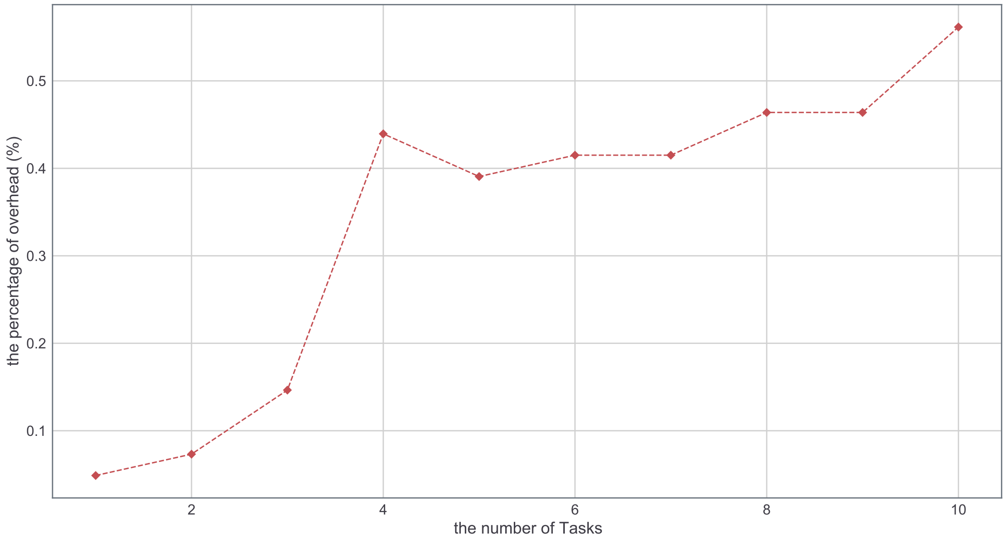


Figure 7: overhead (in percentages %)

2.6 Question6

- **Period**: In event-based scheduling, the minimum period of all tasks is the interruption time of the system. A smaller period represents that the scheduler re-schedules more frequently, resulting in higher overhead. If the minimum period is quite small, the scheduling will degenerate to the tick-based one.
- **Execution Times**: A task set where tasks with higher priorities have longer execution time will be influenced less by scheduling and interruption because there is no need to execute context switch. In contradictory, tasks with lower priorities and longer execution time will be interrupted frequently and the context will be switched for the occurrence of high-priority tasks.

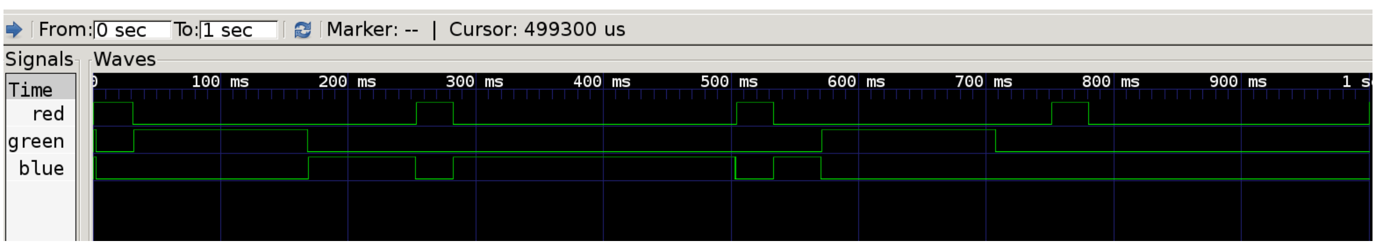
3 Part3

3.1 Question1

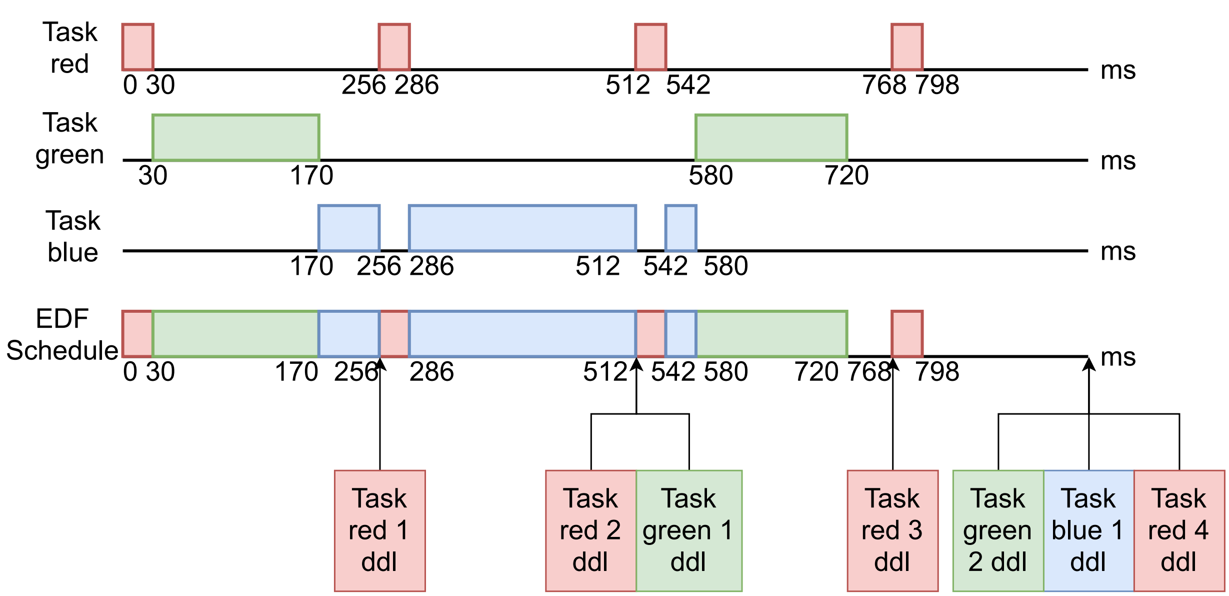
More details are shown in the code.
You can download the code to see the completed version (recommended) or the code in the appendix.

3.2 Question2

Sketch and EDF scheduling results for Tst1, Tst2 and Tst3 can separately be viewed in figure 8, 9 and 10. From this, we can see that the experimental results correspond with our theoretical scheduling in sketches.

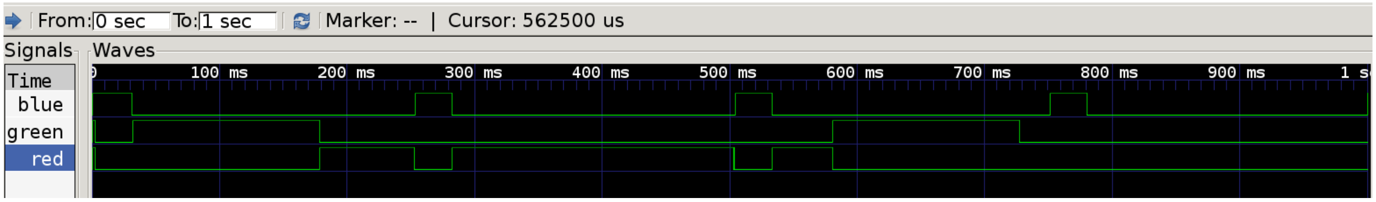


(a) Tst1 Experimental Result

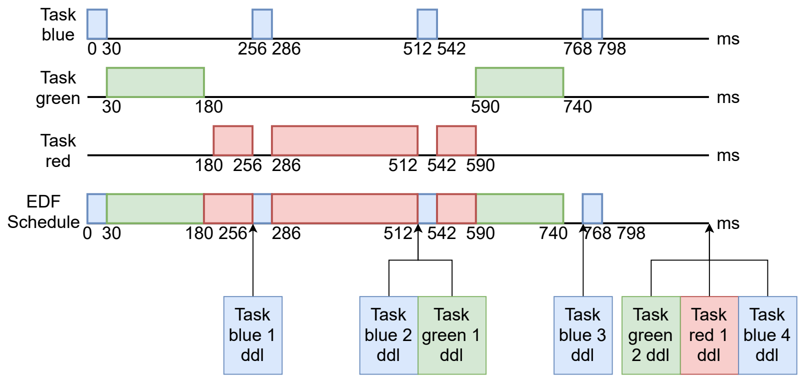


(b) Tst1 Sketch

Figure 8: Comparison of snapshot and sketch for Tst1

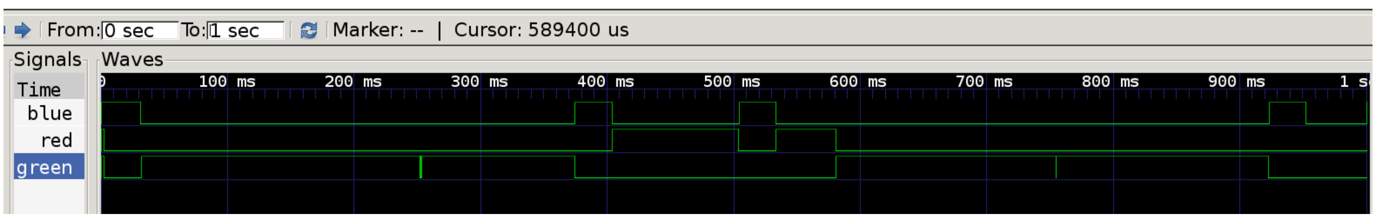


(a) Tst2 Experimental Result

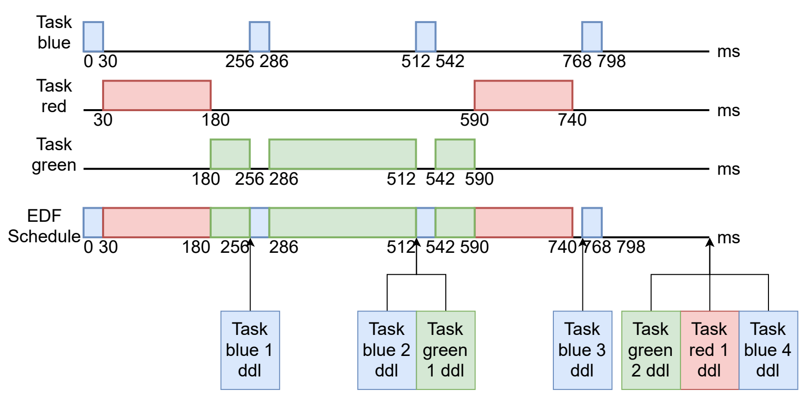


(b) Tst2 Sketch

Figure 9: Comparison of snapshot and sketch for Tst2



(a) Tst3 Experimental Result



(b) Tst3 Sketch

Figure 10: Comparison of snapshot and sketch for Tst3

3.3 Question3

Different from part 2, in order to track the deadline of each task, another parameter $Deadline$ is implemented in the structure **Task**. For this periodic task set, the periods of tasks are the multipliers from each other, and therefore, the minimum interruption period of the tasks is the minimum period of the tasks while the maximum period is the hyperperiod. In each hyperperiod, the server will execute the tasks following the earliest deadline first scheduling. It will first find and execute the task which has the shortest period of time to reach its next release time one by one. At the end of each execution, it will update the deadlines for every task. If the task is completed during that period, its deadline will be set to infinite so as to avoid being selected again. The system will try its best effort to do as many tasks as it can before running out of the remaining time of this hyperperiod - the time slot before the next interruption comes. The complexity of the algorithm is $O(n^2)$. The pseudo-code of algorithm 1 has been shown below and the flowchart has been presented in figure 11.

Algorithm 1: Earliest Deadline First (EDF)

input : $delta \leftarrow \frac{minPeriod}{4}$, task container t, task position $minD$, temple storage $temp$, task number $NUMTASKS$
output : EDF Scheduling

```
for j ← 0 to NUMTASKS do
    temp ← MAX_INT
    for i ← 0 to NUMTASKS do
        if Tasks[i]'s DDL is current minimum then
            temp ← Tasks[i]'s DDL
            minD ← i
        end
    end
    t ← &Tasks[minD]
    if t is pending then
        if enough scheduling time then
            execute all pending tasks of t
            update( t−>Invoked, all tasks ddl, t−>Deadline, delta, t−>Remaining. )
        else
            try best to execute all pending tasks of t
            update( t−>Remaining, all tasks ddl, delta. )
        end
    end
end
```

3.4 Question4

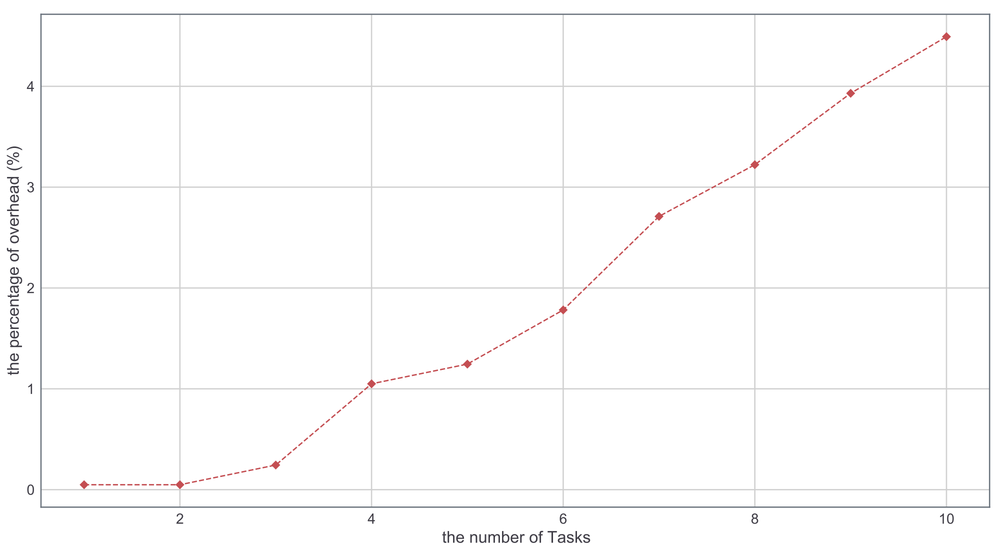


Figure 12: overhead (in percentages %)

According to 12, with the increase of the number of tasks, the average overhead is also growing.

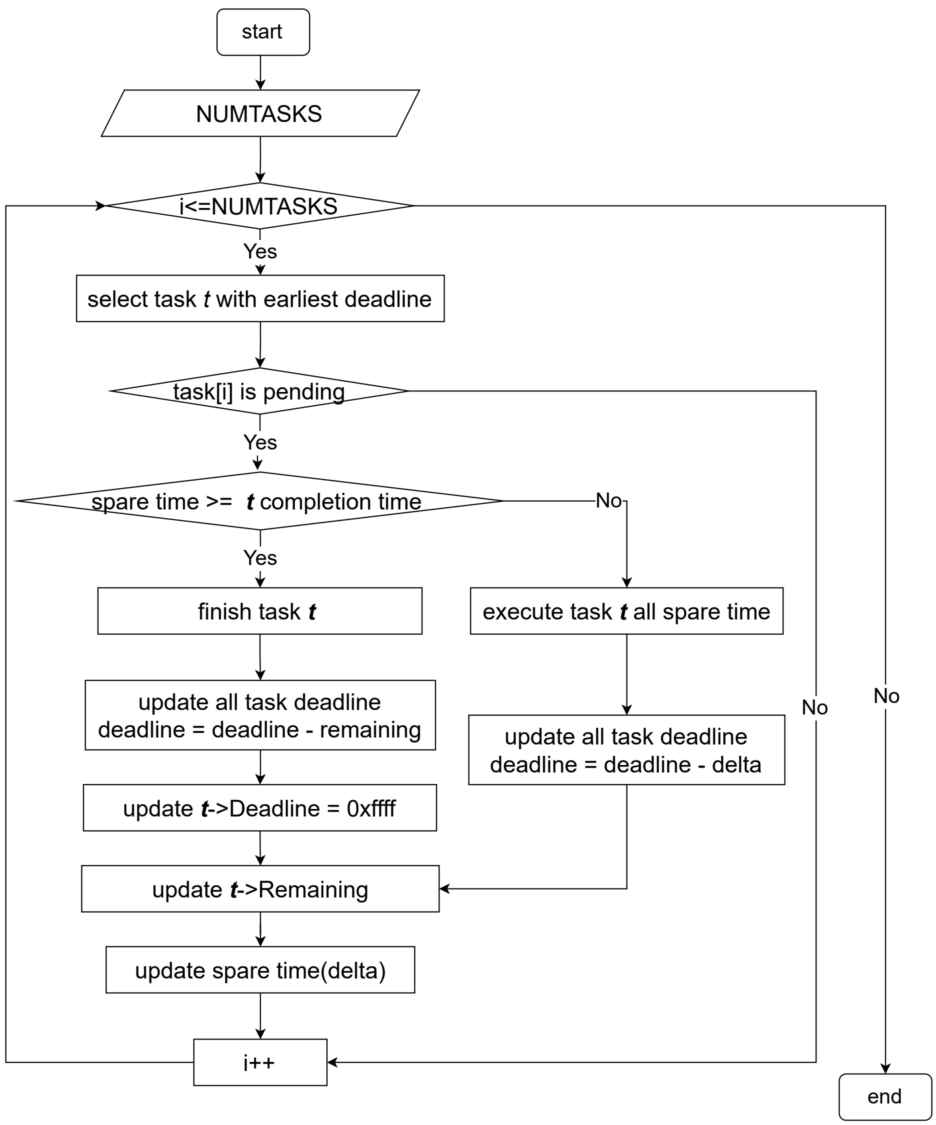


Figure 11: EDF Flowchart

4 Appendix

4.1 Code Used in Part 2

```
interrupt(TIMERA0_VECTOR) TimerIntrpt(void)
{
    ContextSwitch();

    /* ----------------------- INSERT CODE HERE ----------------------- */

    /* Insert timer interrupt logic, what tasks are pending? */
    /* When should the next timer interrupt occur? Note: we only want interrupts at job releases */

    int i;
    for (i = 0; i <= sizeof(Tasks); i++) {
        Taskp t = &Tasks[i];
        if (NextInterruptTime == t->NextRelease) {  // the interruption catches this task's release
            t->NextRelease += t->Period;             // to update the next releasing time of it
            t->Remaining += t->ExecutionTime;        // to calculate the total remaining execution time
            t->Activated++;                          // to add this task to its queue
        }
    }

    NextInterruptTime = (&Tasks[minAt])->NextRelease; // the next timer interrupt is set to the closest release time

  /* ---------------------------------------------------------------- */

    TACCR0 = NextInterruptTime;

    CALL_SCHEDULER;

    ResumeContext();
}
```

```
void Scheduler_P_FP(Task Tasks[]) {
        uint16_t delta = minPeriod / 4;          // spare time for this slot, initialized as the minimum period

        int i;
        for (i = sizeof(Tasks); i >= 0; i--) {   // loop for all the tasks in Tasks[i], ps: Lower indices -> lower priorities

            Taskp t = &Tasks[i];                 // to extract task_i

            if (t->Activated != t->Invoked) {    // if there are task_i(s) pending

                if (delta >= t->Remaining) {     // if the current spare time is enough
                    t->Taskf(t->Remaining);      // to execute all the pending task_i(s)
                    t->Invoked = t->Activated;   // to update "Invoked" to show no more pending
                    delta -= t->Remaining;       // the spare time is consumed
                    t->Remaining = 0;            // since no more pending, no remaining execution time
                }
                else {                           // if the current spare time is insufficient
                    t->Taskf(delta);             // try its best to execute as much time as possible for task_i(s)
                    t->Remaining -= delta;       // to update the remaining execution time
                    //t->Invoked is not [necessary] to be updated here, as there are still remaining tasks in the queue
                    delta = 0;                   // no more spare time, no more tasks can be executed in this slot
                }
            }
        }
    /* ---------------------------------------------------------------- */
```

```
}
```

```
void Scheduler_P_EDF (Task Tasks[])
{
  /* insert code */
  /* Use ExecuteTask function to execute a task */
        //StartTracking(TT_SCHEDULER);

        uint16_t delta = minPeriod/4;                     // spare time for this slot, initialized as the minimum period

        uint8_t i, j, minD;                                                // loop iterators, prepared for the current minimum ddl
        Taskp t;                                                                   // prefared for the current minimum ddl task
        uint16_t temp;                                                     // temple storage initialized as maximum ddl

        for (j = 0; j < NUMTASKS; j++) {                  // loop for all tasks

                /* To select the task owning the cloest ddl*/
                temp = 0xffff;
                for (i = 0; i < NUMTASKS; i++) {
                        if ((&Tasks[i])->Deadline < temp) {
                                temp = (&Tasks[i])->Deadline;   // current minimum ddl
                                minD = i;                                                  // current position where the task has the minimum ddl
                        }
                }
                t = &Tasks[minD];                                          // the selected task
                /*----------------------------------------*/

                if (t->Activated != t->Invoked) {
                        if (delta >= t->Remaining) {                       // if the current spare time is enough
                                //StopTracking(TT_SCHEDULER);
                                t->Taskf(t->Remaining);                    // to execute all the pending task_i(s)
                                //StartTracking(TT_SCHEDULER);
                                t->Invoked = t->Activated;                 // to update "Invoked" to show no more pending
                                for (i = 0; i < NUMTASKS; i++) {           // to update all tasks deadline
                                        (&Tasks[i])->Deadline -= t->Remaining;
                                }
                                t->Deadline = 0xffff;                      // to update the finished task's ddl to infinite,
                                                                                                           //
```

```
                                delta -= t->Remaining;                          // the spare time is consumed
                                t->Remaining = 0;                               // since no more pending, no remaining execution time
                        }
                        else {
                                //StopTracking(TT_SCHEDULER);
                                t->Taskf(delta);                                // try its best to execute as much time as possible for task_i(s)
                                //StartTracking(TT_SCHEDULER);
                                t->Remaining -= delta;                          // to update the remaining execution time
                                for (i = 0; i < NUMTASKS; i++) {        // to update all tasks deadline
                                        (&Tasks[i])->Deadline -= delta;
                                }
                                delta = 0;                                      // no more spare time, no more tasks can be executed in this slot
                        }
                }
        }

  /* insert code */
        //StopTracking(TT_SCHEDULER);
        //PrintResults();
}
```