Reproduction of WRR algorithm based on task QoS

TABLE OF CONTENTS

Abstract

The selected essay [1] proposes a weighted Round Robin (WRR) scheduling algorithm based on Quality of Service (QoS) to handle real-time tasks on lightweight devices. In this paper, we reproduce the WRR algorithm in a popular operating system and analyse its performance. The algorithm utilizes the Token Bucket model (TSPEC) to anticipate the burst time of tasks and calculates different quantum times depending on the priority and QoS requirements of tasks. The algorithm is compared with the traditional Round Robin (RR) scheduling algorithm by simulation and evaluation. The evaluation results show that the algorithm can dramatically reduce the waiting and turnaround time of high-priority tasks, and improve the CPU response rate and resource utilization. Additionally, the paper proposes some improvements to the algorithm to make it adaptable to different task arrival times and total times, which is closer to the real situation.

Chapter 1 Introduction

Unlike common devices, most tasks that lightweight devices need to execute are real-time tasks [2]. This means that lightweight devices have a highly variable task generating rate (sometimes, a burst of tasks are generated in a short time, and sometimes there are no tasks for a long time). As a result, the task queue for RR scheduling can become either too long or too short, leading to a decrease in CPU response rate. To address this issue, it is necessary to regulate the rate of task entry within a certain range. The author introduces a rate control model based on the Token Bucket model [3].

1.1 Token Bucket model

Token Bucket is a container with a specific maximum capacity that holds tokens. Tokens enter the bucket at a specific rate. Once the number of tokens reaches the maximum size of the bucket, new tokens will be discarded. When a task arrives, it can obtain tokens from the Token Bucket directly, and then be transmitted. Hence, the Token Bucket model is like a gate that can control the passage of tasks through tokens.

When a large number of real-time tasks enter within a short period, they are stored in the buffer on the left. The tasks will then pass through the left Token Bucket at a rate of p, which is the peak rate of token entry, since the maximum capacity of the left Token Bucket is 0. After passing through the left Token Bucket, the tasks will enter the second buffer. The tasks will continue to pass at the original rate p until the right Token Bucket reaches empty (0 token). When the Token Bucket is empty, the tasks pass at the average rate r of the right Token Bucket. Therefore, the speed of the tasks passing through Fig. 1 (the red solid line) shows difference, with the first section

being faster than the second section. The task queue can rapidly increase in size and then gradually reach a balance with CPU task execution, while keeping the queue length within a certain range to optimize CPU response rate.

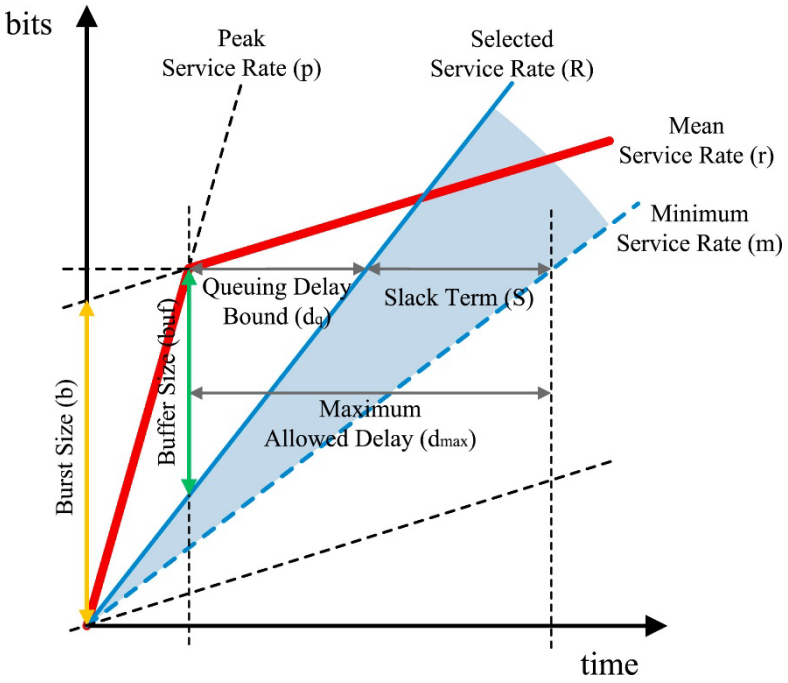

Fig. 1. The parameters and descriptions.

The values of Burst size b and Buffer size buf can be calculated from Fig. 1

$$b = (p - r) \times t$$

$$buf = (p - R) \times t$$

An algebraic substitution can be used to obtain an expression for buf with respect to b.

$$buf = \frac{(p - R)}{(p - r)} \times b$$

The variable d_q indicates the maximum buffer delay. This value can be calculated by dividing the buffer size by the service rate.

$$d_q = \frac{buf}{R} = \frac{(p - R)}{(p - r)} \times \frac{b}{R}$$

The formula for the service rate R can be obtained through mathematical substitution.

$$R = \frac{p}{1 + \mathrm{d}_q \cdot \frac{p - r}{b}}$$

1.2 Formula Derivation

The WRR algorithm is based on Round Robin scheduling (RR) and dynamically adjusts the quantum time of a task based on its priority and different QoS [4] of the task.

The algorithm maintains two separate queues for processing real-time and non-real-time tasks (see Fig. 2). Real-time tasks will be processed first, followed by non-real-time tasks in one round.

The variable d_q represents the theoretical delay in task execution. In reality, however, it is necessary to consider the influence of the round interval ro. Therefore, the actual delay is equal to $dmax + ro$ in the case of the minimum quantum time ratio m (Fig. 1. blue dashed line).

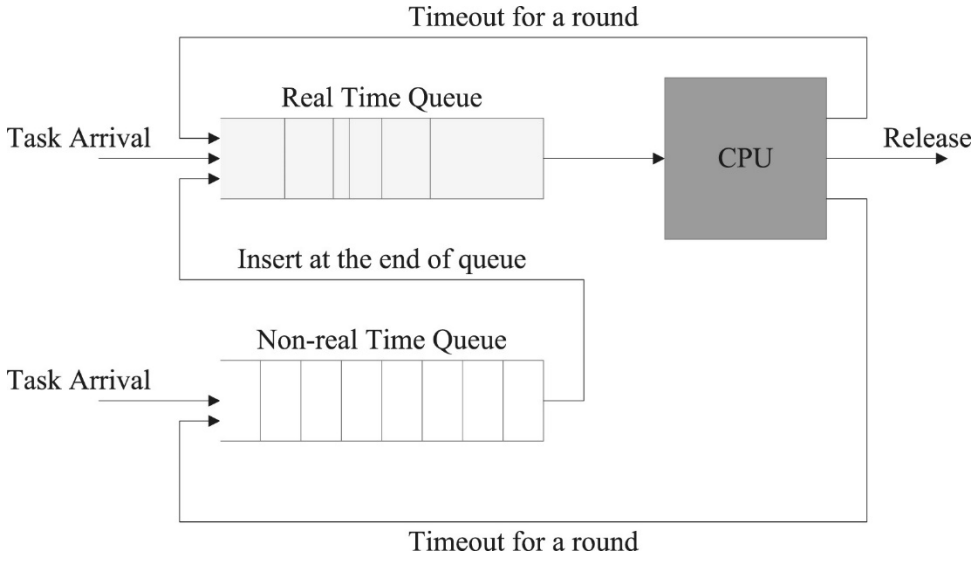

Fig. 2. Layout of the queue in task execution.

The CPU resource slack term S will be set to different ratios S_p based on the maximum allowed delay d_{max} for different tasks, as shown in Table 1.

Table 1
Priority-based slack term rate.

Priority	Application categories	Priority-based slack term rate, S_p
1	Non-real-time application	0%
2	Real-time application	25%
3	Critical real-time application	50%

Thus, we can obtain an expression for the slack term S as:

$$S = (d_{max} + ro) \cdot Sp$$

In order to estimate the rate of the selected quantum time R, the task time should first be parameterized as {r, b, p} as in the Token Bucket model. Table. 2 shows an example of different burst times in a task.

Table 2
Burst time transition of a sample task (in milliseconds).

i	Time (t_i)	Burst time (B_i)	Total burst time ($\sum B_i$)	Average burst time ($\sum B_i/t_i$)
1	10	2	2	0.2
2	20	7	9	0.45
3	30	0	9	0.3
4	40	3	12	0.3
5	50	4	16	0.32
6	60	2	18	0.3

The maximum average burst times is represented by the peak rate p, which is 0.45 in Table. 2.

$$p = max\left(\sum_{i=1}^{N} B_i \Big/ t_i\right)$$

The average rate r is the average burst time after the peak rate p.

$$r = \frac{\sum_{k=t_p+1}^{N} B_k}{t_N - t_p}$$

N is the number of rounds to calculate quantum time, which is 6 here.

t_p is the time of peak rate p, which is 20 here.

t_N is the time of the task round ends, which is 60 here.

The calculated average rate r is 0.225.

The burst size b represents the maximum amount of burst in a round. It is equivalent to the maximum amount of CPU execution time allowed during a task execution round.

$$b = (p - r) \times t_p$$

In this example, the burst size b becomes 4.5.

By integrating the formulas and expressions derived in the Token Bucket model, it is possible to calculate the value of the selected quantum time rate R.

$$d_q = \begin{cases} (d_{max} - S) & if \ d_{max} > S \\ 0 & d_{max} \leq S \end{cases}$$

$$R = \frac{p}{1 + \mathrm{d}_q \cdot \frac{p - r}{b}}$$

The quantum time can be obtained by multiplying the selected quantum time rate R by the round interval ro.

$$quantum\ time = R \times ro$$

We successfully used the Token Bucket model to predict the quantum time of the task.

Chapter 2 Methodology

2.1 Simulation Conditions

Due to the great complexity of using the same platform (embedded board) and methods (design different tasks on TinyOS [5]) to perform experiments as introduced in essay, we turned to adopt simplified simulation using C code to focus on comparing the new algorithm with traditional Round Robin. To better analyze the impact of dynamic quantum time, we allocate the same total time and same executing order for tasks, and the main difference of them is they have different priority or different quantum time in our experiment data set. For traditional Round Robin, all tasks will be allocated the same quantum time given by system, so we set a constant static number as quantum time for RR. On the contrary, Weighted Round Robin based on Token Bucket model will first give different slack term rate Sp according to priority of tasks, and then calculate the specific quantum time based on Sp and other parameters derived from the parameterization of the Token Bucket model.

2.2 Reproduction

As conclusion, the main tasks of our implementation are calculating all required parameters from the data set we design, then prepare a sorting function to set the same tasks executing order of tasks, and finally write RR and WRR scheduling functions to simulate the executing process of tasks for further performance analysis. The overall structure of the reproduction is shown in Program. 1.

Program 1 GetQuantum and Evaluate Procedures

Input: FILE: Testing file contains tasks' ID, burst time, dmax, ro and prority, also the number of tasks
Output: T: Tasklist with waiting time and turnaround time

```
1: function GetQuantum(FILE)
2:     Tasklist T < ID, priority... > ← HandleFile(FILE)
3:     T < other members > ← ∅
4:     T < p, r, b > ← Parametrize(T)
5:     CalculateQuantum(T) ← CalculateDelay(T)&Calculate_R(T)
6:     CalculateQuamtum(T)
7: end function
8:
9: function Evaluate(T)
10:    RR(T) ← SortTasks(T)
11:    T < waiting time, turnaround time > ← RR(T)
12:    WRR(T) ← SortTasks(T)
13:    T < waiting time, turnaround time > ← WRR(T)
14: end function
```

The code divides our reproduction into two parts, initialize the tasks and put them into scheduling process. In initialize process, we use structure pointer to store the parameters and other properties of tasks, which will be read from our testing file. Then, according to the given tasks number in file, tasks will be allocated space dynamically. After successfully allocating space, the data in file will be written to list while other members will be initialized to be written later. Coming to the next part, the parameters needed to calculate quantum time will be derived through parametrization and existing formulas we get in former chapter. Finally, it's the most important part of our reproduction, sort tasks and scheduling. In sorting function, we use *qsort()* to ensure tasks with higher priority or longer quantum time be placed first, as shown in Algorithm. 1 [6].

Algorithm 1 Sorting by priority

Input: T
Output: ∅

```
1: function SortTasks(T)
2:     function cmp(T1, T2)
3:         if T2.priority > T1.priority then
4:             return 1
5:         else if T2.quantum > T1.quantum then
6:             return 1
7:         else
8:             return 0
9:         end if
10:    end function
11:    qsort(T, cmp)
12: end function
```

Both RR and WRR will execute tasks in this order so we can focus on comparing static and dynamic quantum time. In the last part, we will simulate RR and WRR scheduling and display the waiting time and turnaround time of each task for later evaluation. Algorithm. 2 [6] illustrates the scheduling process.

Algorithm 2 WRR Scheduling

```
   Input: T
   Output: ∅
1: function SCHEDULING(T)
2:     SortTasks(T)
3:     while not all tasks finished do
4:         if task won't be finished in this turn then
5:             left ← left time − T < quantum >                    ▷ update left time
6:             if not first turn then
7:                 waiting time ← waiting time + (start − end)      ▷ update waiting time
8:             end if
9:             end ← start + quantum                               ▷ update end time of current task
10:            T_next < start >← T < end >                         ▷ update start time of next task
11:            if reach end of turn then
12:                T_start < start >← T < end >                    ▷ start from beginning
13:            end if
14:        else
15:            left ← left time − T < quantum >                    ▷ update left time
16:            if not first turn then
17:                waiting time ← waiting time + (start − end)     ▷ update waiting time
18:            end if
19:            end ← start + quantum                               ▷ update end time of current task
20:            T_next < start >← T < end >                         ▷ update start time of next task
21:            if reach end of turn then
22:                T_start < start >← T < end >                    ▷ start from beginning
23:            end if
24:            left ← end                                          ▷ record turnaround time
25:            Start from next task
26:        end if
27:    end while
28: end function
```

When a task enqueue, we need to judge whether it will be finished in this turn by comparing its left time and quantum time. If left time is larger than quantum time, it won't be finished; else, it will. In both situations, we need to update left time, task's end time, waiting time (start time this turn – end time last turn) and next task's start time (end time this turn). When a turn ends, start from beginning. And when a task is finished, set left time as 0 and record turnaround time as end time. Since we have sorted tasks before, tasks with longer quantum will be executed first in WRR so tasks will be finished by order. Thus, when a task finishes, we simply start from the next.

2.3 Possible Improvement

The previously designed algorithms fixed the $Totaltime$ of each task to 30 and the $Arrivaltime$ to 0. However, in practice it is not possible to fix the $Totaltime$ of a task, while each task will have a different $Arrivaltime$, hence we redesigned the entire algorithm section to account for these variations.

The formulas for calculating r and b were also refactored, dividing the $Totaltime$ into $TASK\_ROUND$ parts and recalculating their sizes. Additionally, in the task lists, each task is sorted by $Arrivaltime$ after initialization to ensure that tasks with small $Arrivaltime$s will be executed first. If two tasks have the same $Arrivaltime$, they will be sorted by priority. The algorithm is illustrated in Algorithm. 3 [6].

Algorithm 3 Reconstructed Algorithmic Structures

Input: ∅
Output: ∅

```
function ORGANIZE                                   ▷ Organize the task by arrival time or priority
    SortTasks(Tasklist, Listindex, cmp)             ▷ sort tasks(tail index is Listindex) by cmp function
end function

function CMP(T1, T2)                                ▷ compare two tasks by arrival time
    Sort by arrival time.
    if T1.arrivalTime > T2.arrivalTime then         ▷ if T1 arrives later than T2
        return 1
    else if T1.arrivalTime < T2.arrivalTime then    ▷ if T1 arrives earlier than T2
        return -1
    else                                            ▷ if T1 and T2 have the same arrival time
        return cmp_priority(T1, T2)                 ▷ sort by priority in Algorithm. 1
    end if
end function
```

As shown in Algorithm. 4 [6], the optimized WRR algorithm initializes a clock with the smallest $Arrivaltime$ of all tasks before which no tasks will be executed. During each round of scheduling, the cost of each task should be calculated as shown in Algorithm. 5 [6]. Check if the current task has arrived and if it has not finished yet, execute it for quantum time. Otherwise, execute it for the left time. Then, check if any unarrived tasks have arrived at the time of execution and increase the waiting time of other arrived tasks. At the end of each round, the algorithm checks for newly arrived

tasks and resorts the list based on priority. The scheduling algorithm will terminate when all tasks have been executed.

This optimized WRR algorithm solves the problem of unknown total time and arrival time, while it is more adaptable to handle realistic conditions with different tasks.

Algorithm 4 Optimized WRR algorithm

```
    Input: ∅
    Output: ∅
 1: function WRR_SCHEDULING_EDITED
 2:     clock ← Tasklist[1].arrivalTime                ▷ Initialize clock to the first task's arrival time
 3:     round ← 1                                      ▷ Initialize round to 1
 4:     while TRUE do
 5:         isAllFinish ← 0                            ▷ Counting the tasks accomplished
 6:         resort ← 0                                 ▷ Used to determine if reordering is required
 7:         round ← round + 1
 8:         for i from 1 to Listindex do
 9:             backupClock ← clock                    ▷ store the current clock
10:             if Tasklist[i].arrivalTime > clock and Tasklist[i].arrivalFlag == 0 then
11:                 continue                           ▷ if the task has not arrived yet, skip it
12:             end if
13:             waitTime ← 0.00                        ▷ set waitTime increment for other tasks
14:             TaskRun(Tasklist[i], clock, waitTime)  ▷ run the task
15:             for j from i + 1 to Listindex do
16:                 if Tasklist[j].arrivalTime ≤ clock and Tasklist[j].arrivalFlag == 0 then  ▷ arrived
17:                     Tasklist[j].arrivalFlag ← 1    ▷ set the arrivalFlag to 1
18:                     resort ← j                     ▷ set resort to the task index j
19:                 end if
20:             end for
21:             for j from 1 to Listindex do           ▷ update the waiting time of other tasks
22:                 if i ≠ j and Tasklist[j].isFinish == 0 and Tasklist[j].arrivalFlag == 1 then
23:                                                    ▷ the other task is not finished and has arrived
24:                     Tasklist[j].waitTime ← Tasklist[j].waitTime + waitTime
25:                 end if
26:                 if Tasklist[j].arrivalTime ≤ clock and Tasklist[j].arrivalTime ≥ backupClock then
27:                                                    ▷ if the task has arrived in this round
28:                     Tasklist[j].waitTime ← Tasklist[j].waitTime − Tasklist[j].arrivalTime
29:                                                    ▷ update the waitTime
30:                 end if
31:             end for
32:             if Tasklist[i].isFinish == 1 then      ▷ if the task is finished
33:                 isAllFinish ← isAllFinish + 1      ▷ increment isAllFinish by 1
34:             end if
35:         end for
36:         if resort > 0 then                         ▷ Tasklist need to reorder by priority
37:             SortTasks(Tasklist, resort, cmp_priority)  ▷ sort tasks by cmp_priority function
38:             resort ← 0                             ▷ set resort back to 0
39:         end if
40:         if isAllFinish == Listindex then           ▷ if all tasks are finished
41:             break                                  ▷ exit
42:         end if
43:     end while
44: end function
```

Algorithm 5 Task Run

Input: Task, current clock and waiting time increment
Output: $\emptyset$

1: **function** TaskRun(T, clock, waitTime_increment)
2: **if** $T < isFinish >$ **then** ▷ if the task is finished
3: **return** ▷ exit
4: **end if**
5: **Executed and calculate the cost.**
6: **if** $T.totalTime > T.quantumTime$ **then** ▷ task is not finished yet
7: $T.totalTime \leftarrow T.totalTime - T.quantumTime$ ▷ update left time
8: $clock \leftarrow clock + T.quantumTime$ ▷ update the clock
9: $waitTime\_increment \leftarrow T.quantumTime$ ▷ set the waitTime to the quantum time
10: **else** ▷ task will finish
11: $T < isFinish > \leftarrow 1$ ▷ set the isFinish to 1
12: $clock \leftarrow clock + T < totalTime >$ ▷ update the clock by adding the left time
13: $T < turnaroundTime > \leftarrow clock - T.arrivalTime$ ▷ calculate turn around time
14: $waitTime\_increment \leftarrow T < totalTime >$ ▷ set the waitTime to the left time
15: **end if**
16: **end function**

Chapter 3 Results and Analysis

In this evaluation, we assume prior knowledge of the $Totaltime$, $Arrivaltime$, and $Burst\ time$ of the tasks to reduce variables, which allows us to observe its optimization for the RR algorithm more clearly.

Consider three tasks with different priorities, shown in Table. 3, Table. 4 and Table. 5. All tasks have the same $dmax$ and ro. After setting $dmax$ and ro. to 10 and computing the two algorithms, we obtained the bar chart shown in Fig. 3. The traditional RR algorithm does not reflect the priority advantage since the waiting time and turnaround time of the three tasks are not significantly different. The WRR algorithm, on the other hand, dramatically reduces the waiting time and turnaround time of the higher priority task (task2), ensuring better QoS. However, this effect is not obvious for the lower priority task (task3).

Table 3
Task 1 with priority 2

i	Time (ti)	Burst time (Bi)	Total burst time(∑Bi)	Average burst time (∑Bi/ti)
1	10	5	5	0.5
2	20	7	12	0.6
3	30	3	15	0.5

Table 4
Task 2 with priority 3

i	Time (ti)	Burst time (Bi)	Total burst time(∑Bi)	Average burst time (∑Bi/ti)
1	10	8	8	0.8
2	20	4	12	0.6
3	30	2	14	0.4666

Table 5
Task 3 with priority 1

i	Time (ti)	Burst time (Bi)	Total burst time(∑Bi)	Average burst time (∑Bi/ti)
1	10	2	2	0.2
2	20	6	8	0.4
3	30	1	9	0.3

Table 6
Task 1 with different *dmax*

dmax	Quantum Time	Waiting Time	Turnaround Time
10	4.8	40.153846	70.153846
20	3.692308	54	84
30	3	60	90

Table 7
Task 1 with different ro

ro	Quantum Time	Waiting Time	Turnaround Time
10	4.8	60	90
20	10.6667	39.6	69.6
30	18	15.466667	45.466667

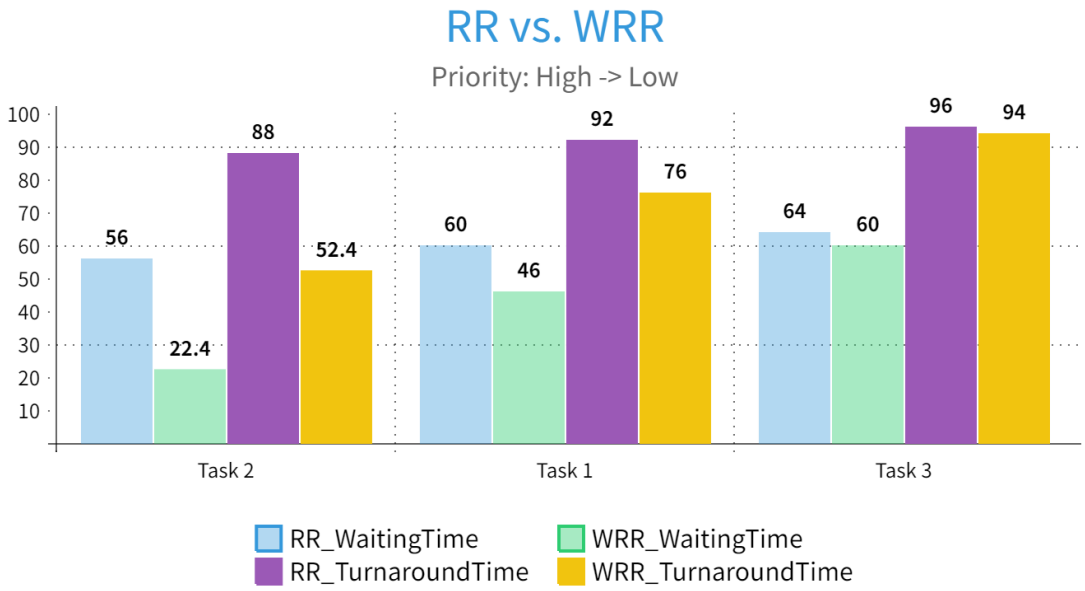

Fig. 3. Comparison of RR and WRR algorithms for different priority tasks.

Now consider the influence of different $dmax$ and ro on the experimental results. With all other variables being equal, as in Table. 6 and Fig. 4, an increase in $dmax$ leads to a decrease in quantum time, whereby the tasks will be sorted backwards. As shown in Table 7 and Fig. 5 As the value of ro increases, the quantum time also rises, and tasks with longer intervals will be executed first, which can make the scheduling more efficient.

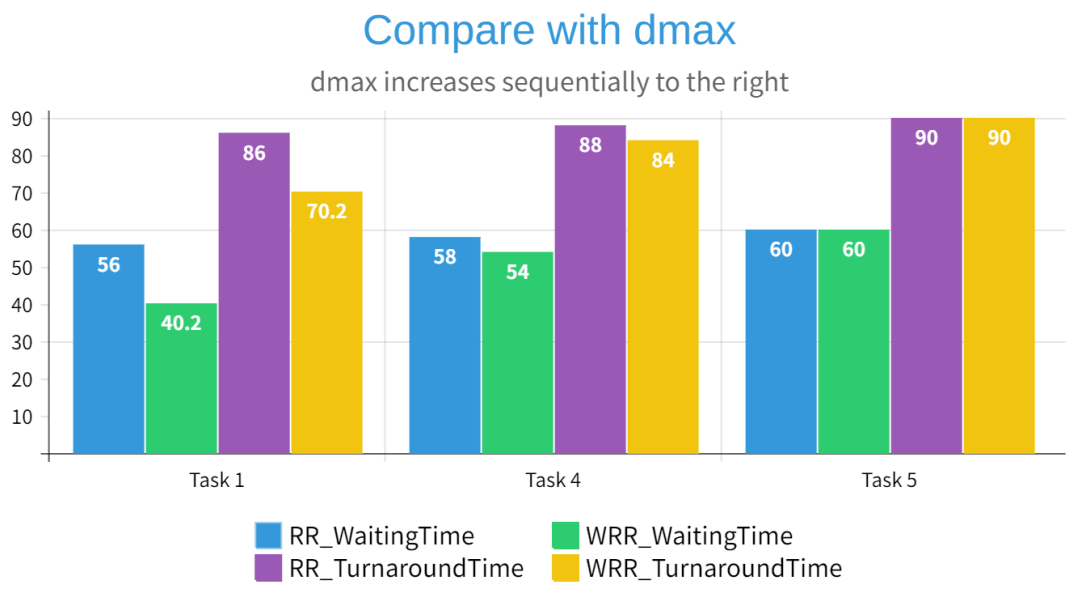

Fig. 4. Comparison plots of the two algorithms at various $dmax$ values.

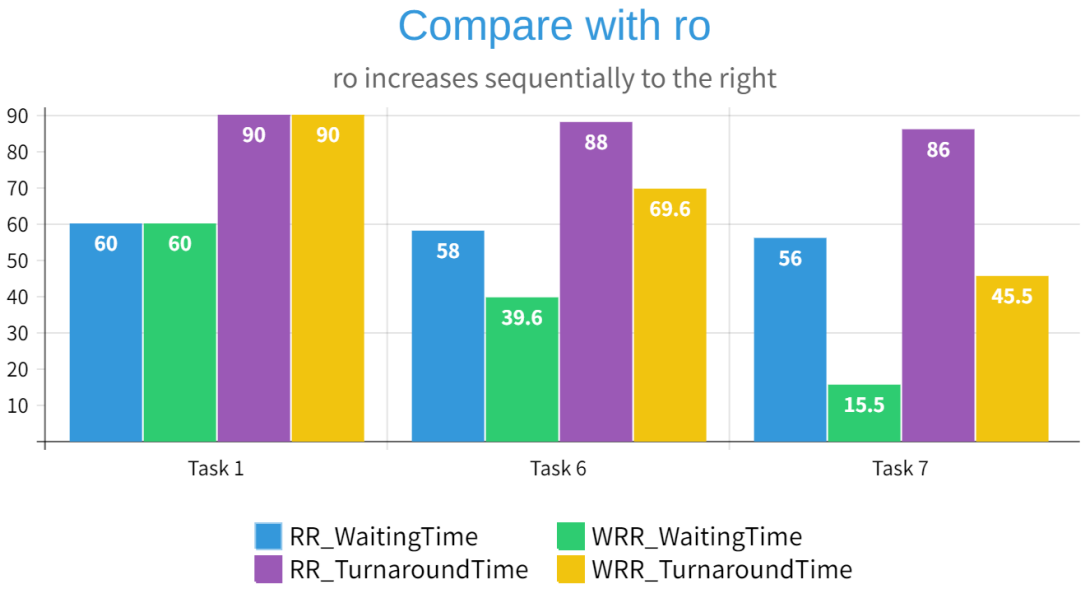

Fig. 5. Figure comparing the two algorithms under different ro conditions.

Chapter 4 Conclusions

In short, the WRR algorithm is an optimized RR scheduling algorithm that allocates CPU resources unevenly based on the QoS requirements of tasks. The Token Bucket model is utilized to estimate the burst time of tasks, and to calculate different quantum times for each task. The WRR algorithm can significantly reduce the waiting time and turnaround time of high priority tasks through evaluation. In the future, this algorithm is expected to play an important role in lightweight devices.

Reference

[1]: Kim, S. (2017b). QoS provisioning of a task-scheduling algorithm for lightweight devices. *Journal of Parallel and Distributed Computing*, *107*, 67–75. https://doi.org/10.1016/j.jpdc.2017.04.010

[2]: Leung, J. Y., & Whitehead, J. (1982). On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, *2*(4), 237–250. https://doi.org/10.1016/0166-5316(82)90024-4

[3]: Huawei Support. (2021). Traffic metering and token bucket mechanism. https://support.huawei.com/enterprise/en/doc/EDOC1000178175/f6e567c8/traffic-metering-and-token-bucket-mechanism

[4]: Huawei Support. (2022). What Is Quality of Service (QoS)? https://support.huawei.com/enterprise/en/doc/EDOC1100086518

[5]: Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Hill, J., Welsh, M., Brewer, E., & Culler, D. (2005). TinyOS: an operating system for sensor networks. In *Springer eBooks* (pp. 115–148). https://doi.org/10.1007/3-540-27139-2_7

[6]: Hoyue. (2023). Reproduction of WRR algorithm and improvement. https://github.com/Aki-Hoyue/projects/tree/main/OS_lab/WRR