IN4343 REAL-TIME SYSTEM LAB2 - GROUP 24

Yuhang Tian
5219728

Mingyu Gao
5216281

March 20, 2021

1 Question 1

1.1 SCHED_FIFO

SCHED_FIFO SCHED_FIFO (First in-first out scheduling) is one of the real-time policies. It shows the following features:

- When preempting other threads: The threads under SCHED_FIFO should have static priorities larger than 0, which means that SCHED_FIFO threads can preempt any other normal scheduling threads including SCHED_OTHER, SCHED_BATCH, and SCHED_IDLE.
- When being preempted by other threads: SCHED_FIFO thread can be preempted by other threads with higher priorities and it will resume execution after all preempting threads - threads with higher priorities - finish their executions.
- When running enable: SCHED_FIFO thread will be arranged in the execution list according to its priority.
- When priority changed: The position of the thread in the list will also change according to the new priority.

1.2 SCHED_RR

SCHED_RR threads almost have identical properties compared to SCHED_FIFO, but there are still some differences:

- Every thread is assigned a time quantum. The thread can only be executed in its own time quantum one by one and round by round.
- If a SCHED_RR thread is preempted by another thread, it can only execute its remaining time quantum after being resumed.

1.3 SCHED_DEADLINE

This scheduling is applied in sporadic task model, which contains a sequence of jobs that is only activated once per period. This policy utilizes GEDF(Global Earliest Deadline First) to schedule tasks.

- **Setting**
  - Setting a SCHED_DEADLINE policy for a thread needs the API **sched_setattr** with three parameters, *Runtime*, *Deadline* and *Period*, which correspond to the key features of each job in sporadic task mode.
  - *Runtime* is the time to finish this thread, including but not merely including competition time.
  - *Deadline* is the relative deadline, before which the job should finish execution. This is different from the absolute deadline, obtained by adding the relative deadline to the arrival time.
  - *Period* is the arrival time difference between two tasks.
  - Kernel requires that *sched_runtime* <= *sched_deadline* <= *sched_period*.
- **Priority**
  This policy does not set priority as SCHED_DEADLINE threads always have the highest priority, which means a runnable SCHED_DEADLINE thread could preempt any current running thread using other policy.

2 Question 2

2.1 SCHED_FIFO

The predefined properties of FIFO are shown in table 1, and the result is presented in figure 1. We get the response time for each task shown in table 2.

Table 1: FIFO Properties

Properties	Thread 0	Thread 1	Thread 2	Thread 3
WCET (ms)	5.1	5.1	5.1	5.1
Period (ms)	50	50	50	50
Priorities	1	3	2	4
Utilization		≈ 0.4		

According to figure 1, all the threads release at the same time. Due to the property of FIFO - the higher-priority thread runs first, thread_3 (priority=4) runs earliest followed by thread_1 (priority=3), thread_2 (priority=2) and thread_0 (priority=1).

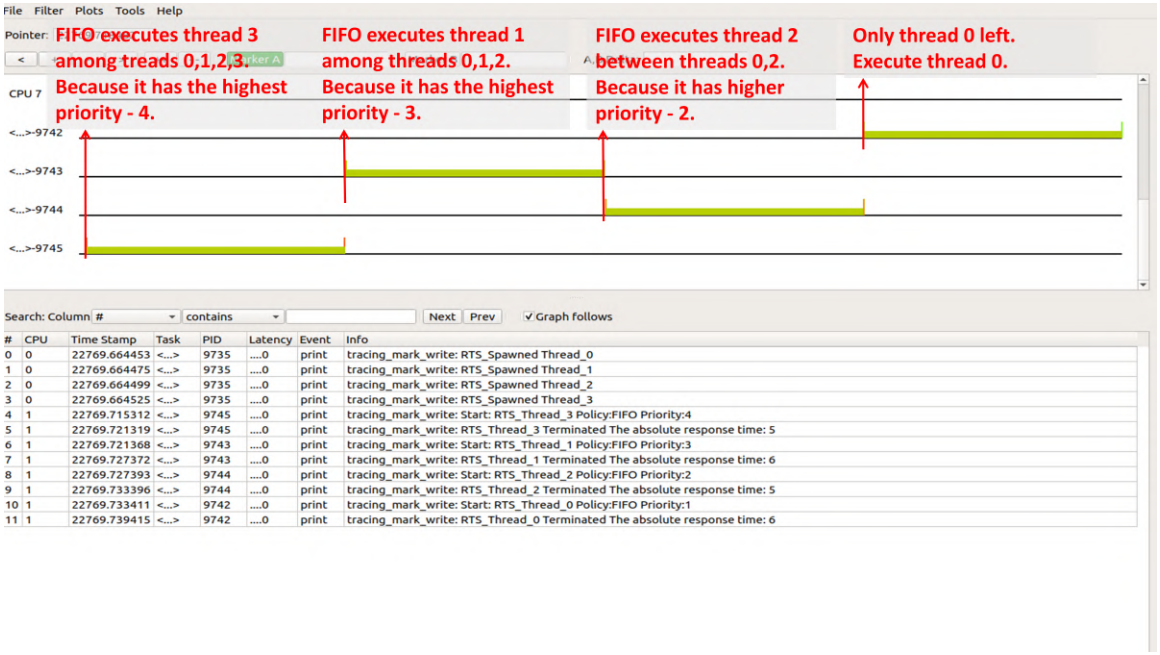


Figure 1: FIFO Simulation

Table 2: FIFO Response Time

Properties	Thread 0	Thread 1	Thread 2	Thread 3
Response Time (ms)	22	11	16	5

2.2 SCHED_RR

The predefined properties of RR are shown in table 3, and the result is presented in figure 2. We get the response time for each task shown in table 4.

According to figure 1, all the threads release at the same time. The RR divide the task execution period into several smaller time quantums. For a task with higher priority like thread_0, although its execution period is broken into slots, it will run till it finishes itself, during which other tasks with lower priority cannot interrupt it. For two tasks with identical priorities like thread_1 and thread_2, they will run in turn and switch to another when each of them runs a specific duration defined by the system. For a single task with a lower priority like thread_3, it will execute consecutively with all its slots when there is no pending task with a higher priority.

Table 3: RR Properties

Properties	Thread 0	Thread 1	Thread 2	Thread 3
WCET (ms)	5.1	5.1	5.1	5.1
Period (ms)	50	50	50	50
Priorities	4	2	2	1
Utilization		≈ 0.4		
Time Slice (ms)		4		
Busy Wait Time (ms)		6		

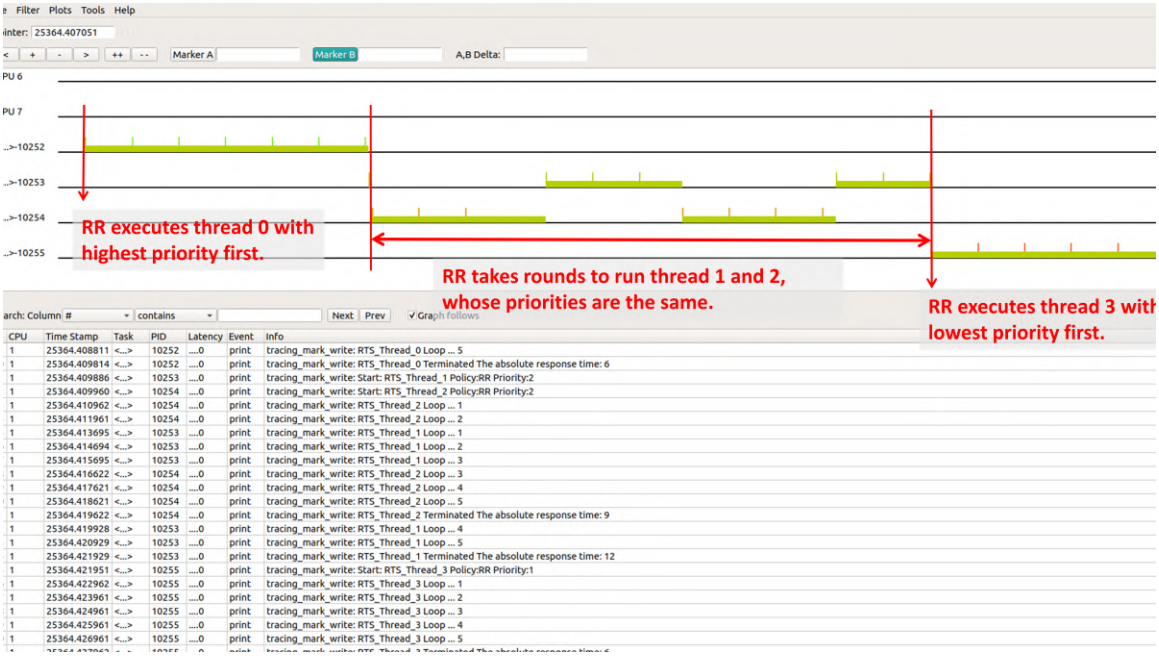


Figure 2: RR Simulation

Table 4: RR Response Time

Properties	Thread 0	Thread 1	Thread 2	Thread 3
Response Time (ms)	6	18	15	24

2.3 SCHED_OTHER

The predefined properties of OTHER are shown in table 5, and the result is presented in figure 3. We get the response time for each task shown in table 6.

Table 5: OTHER Properties

Properties	Thread 0	Thread 1	Thread 2	Thread 3
WCET (ms)	5.1	5.1	5.1	5.1
Period (ms)	50	50	50	50
Priorities	0	0	0	0
Utilization		≈ 0.4		

SCHED_OTHER is the standard Linux time-sharing scheduler, under which the thread to run is chosen from a static priority 0 list based on a dynamic priority that is determined only inside this list. The dynamic priority is based on the nice value. In other words, a task that has been executed for a long while will have a lower dynamic priority, so it is more probably interrupted by other higher-dynamic-priority tasks which have not been executed for a couple of times.

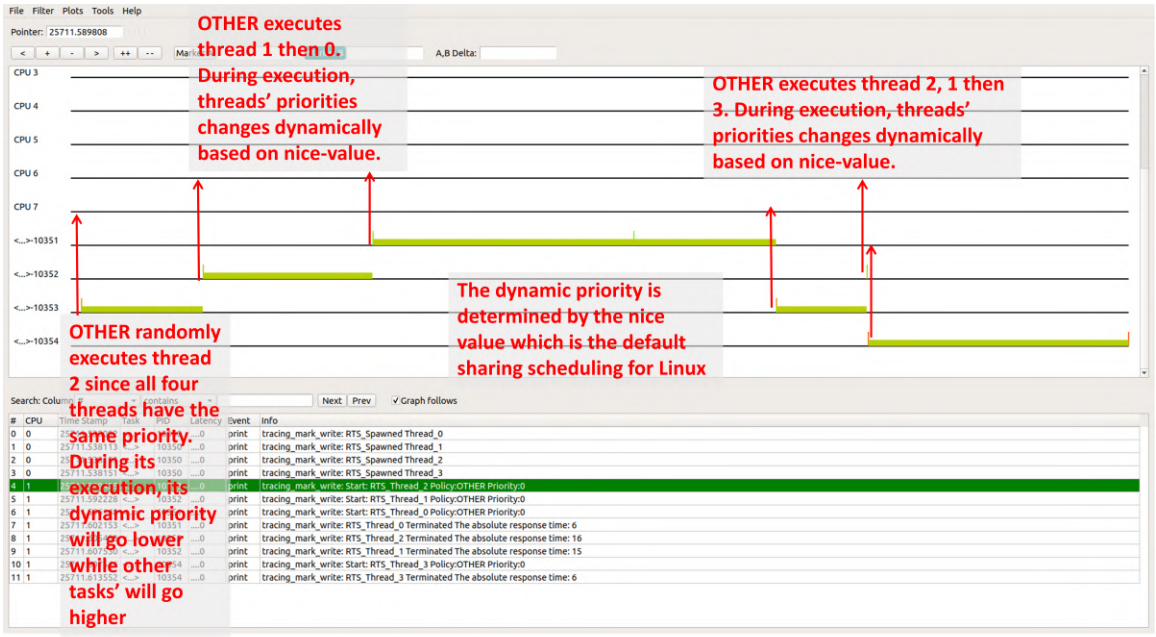
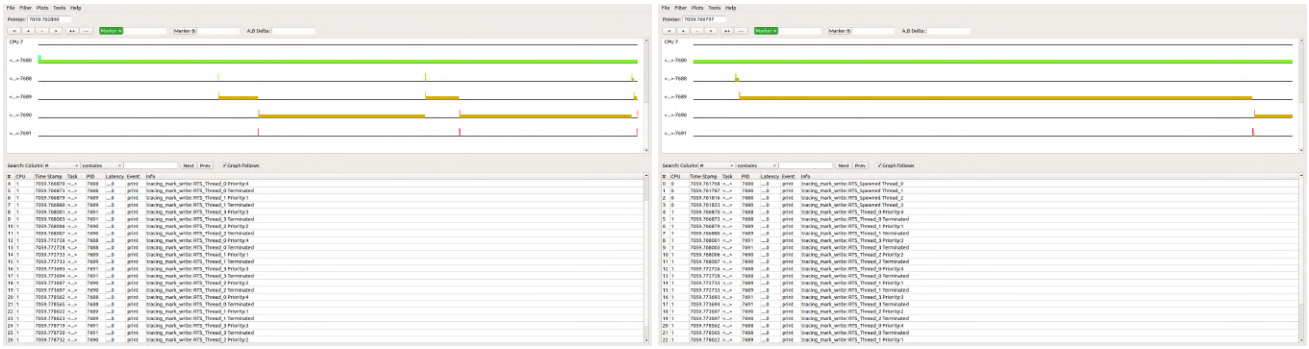


Figure 3: OTHER Simulation

Table 6: OTHER Response Time

Properties	Thread 0	Thread 1	Thread 2	Thread 3
Response Time (ms)	12	18	16	24

3 Question 3

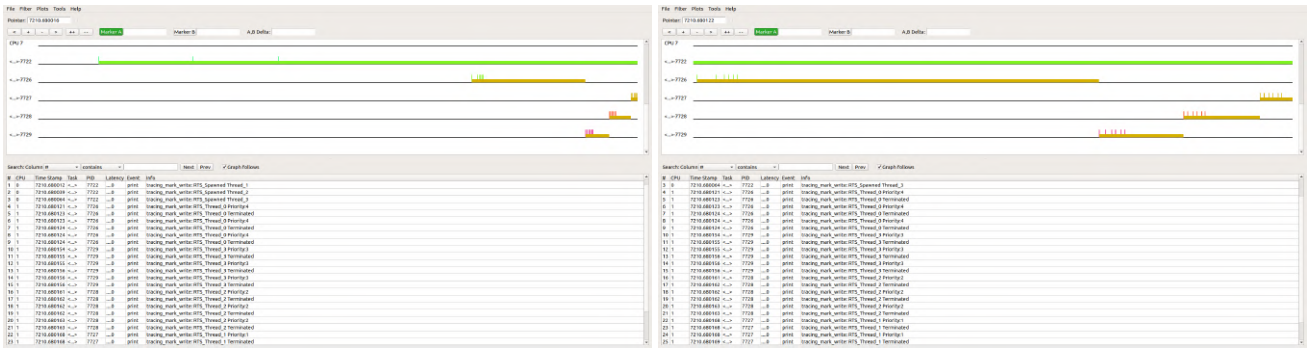


(a) Method 1 - overview (b) Method 1 - Zoom In

Figure 4: Method I

```
interval.tv_sec=0;
interval.tv_nsec = 5e6;
for (int i = 0; i < 3; i++) {
        nanosleep(&interval, 0);
        trace_write("RTS_Thread_%d␣Priority:%d\n",
        args->thread_number,
        args->thread_priority);
        trace_write("RTS_Thread_%d␣Terminated", args->thread_number);
}
```

According to figure 4, for the first method, it can even not distinguish the priorities of the tasks. The tasks betray the rule of executing priority. The difference between the expected arrival times of the task and the actual release times is about 5.30 ms.

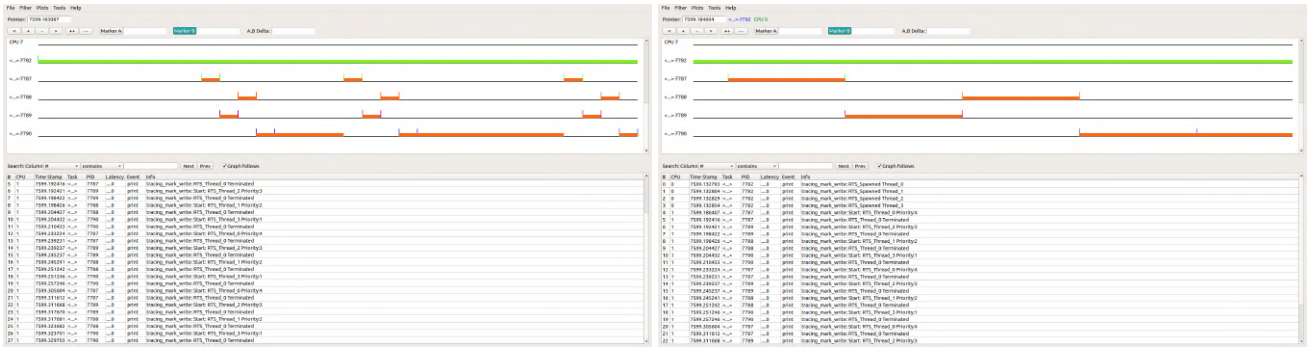


(a) Method 2 - overview (b) Method 2 - Zoom In

Figure 5: Method II

```
for (int i = 0; i < 3; i++) {
        timespec_add_us(&next, interval_us);
    clock_gettime(CLOCK_REALTIME, &now);
    rem.tv_sec = next.tv_sec - now.tv_sec;
    rem.tv_nsec = next.tv_nsec - now.tv_nsec;
    nanosleep(&rem, 0);
    trace_write("RTS_Thread_%d␣Priority:%d\n",
        args->thread_number,
    args->thread_priority);
    trace_write("RTS_Thread_%d␣Terminated", args->thread_number);
}
```

According to figure 5, for the second method, it can identify which task should be executed according to its priority, but it keeps all periods of a task and executes them at once. The difference between the expected arrival times of the task and the actual release times is about 0.12 ms.



(a) Method 3 - overview (b) Method 3 - Zoom In

Figure 6: Method III

According to figure 6, the third method can execute the tasks correctly. All tasks are executed periodic and follow predefined priorities. Besides, all the execution time are accurate as expected. The difference between the expected arrival times of the task and the actual release times is about 53.74 ms. Only the last leads to the periodic actuation, as the thread is preempted between calls to `clock_gettime()` and `!nanosleep()` such that the interval is not correctly computed whereas `clock_nanosleep()` can atomically perform the operations and suspend the tasks with the correct interval.

```
clock_gettime(CLOCK_REALTIME, &next);
for(int i = 0; i < 3; i++){
        timespec_add_us(&next, args->thread_period);
        clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &next, NULL);
        trace_write("Start: RTS_Thread_%d Priority:%d\n",
        args->thread_number,
    args->thread_priority);
        workload(args->thread_number);
        trace_write("RTS_Thread_%d Terminated",results[tid].thread_number);
}
```