Linux Real-Time Schedulers Profiling

Vincent Chen
Master of Computer Science
NJIT
Newark, New Jersey
vc342@njit.edu

Yanxinwen Li
Master of Computer Science
NJIT
Newark, New Jersey
yl2235@njit.edu

***Abstract*—This final project will design different real-time task sets to evaluate schedulers. And using Linux system call tracer strace and Linux kernel tracer ftrace via two utilities called trace-cmd and kernelshark to compare the SCHED_FIFO, SCHED_RR and SCHED_DEADLINE.**

I. Introduction

The difference between real time system and not real time system is that the real time system has real time computing, which means that the real time system need to guarantee that every task will be done before task deadline. For this reason, the real time system need different scheduler algorithm to find the best way to avoid meeting deadline.

Linux real tine scheduling include SCHED_FIFO, SCHED_RR and SCHED_DEADLINE. In this project, we will learn how to trace Linux threads, design different task to understanding the scheduling algorithm and calculate the time of overhead.

II. Milestone

A. *Creating and tracing simple thread*

First, we have to figure out how to create threads with different priority and policy in Linux. Next, using trace-cmd record and kernelshark to trace the task time stamp and event.

B. *Compare different deadline situation*

Create different task sets to compare scheduling algorithm used in real-time operating systems. The three scheduling algorithm is: Rate monotonic scheduling(RM), Earliest deadline first scheduling(EDF) and Round robin(RR).

C. *Third milestone*

We calculate overhead from the scheduling algorithms in milestone 2. We use the data from kernelshark to find out all the time delay between sched_switch and sched_process_exit in whole process, and add them up to compare the efficacy of different scheduling algorithm.

III. Milestone Detail

Here, we will introduce the detail of each milestone.

A. *Creating and tracing simple thread*

In the creating thread code, we use pthread_create() function to create threads, and pthread_join() to make sure the thread is executed. We set pthread_mutex() and pthread_cond() in every thread, to make sure that every task will start at the same time. We think it is a better way to measure task running time. The function sched_setattr() can set scheduler attribute with different setting. We can change policy (FIFO, RR, DEADLINE) and priority (0~99) in sched_policy and sched_priority. We can also set runtime, deadline and period in SCHED_DEADLINE under nano-second. We also use CPU_ZERO(), CPU_SET() and pthread_setaffinity_np() to set every thread in specific cpu. The code will submit in code submission.

For the start, we need to install trace-cmd and kernelshark in our linux system. Next, we can input linux command lines, as in:

ggc *filename.c* –o *executefilename* -lpthread (1)

sudo trace-cmd record –e sched –o *filename.dat ./filename* (2)

kernelshark –i *filename.dat* (3)

The first command can compile thread file and output the file. The second command can record the file with sched_ event (include switch, waking, migrate, process_exit……etc) and output the trace data. The third command can execute kernelshark with reading specific tracing data.

B. *Compare different deadline situation*

This session will introduce the three scheduling algorithm: Rate monotonic scheduling (RM), Earliest deadline first scheduling (EDF) and Round robin (RR).

- RM is the best scheduling algorithm in single core, it was establish by Liu and Layland in 1973. One of the feature in this algorithm is that the utilization factor is easy to implement, high performance and simple. The main spirit is that the priority of task has inverse ratio with task period. Which means that the shorter of task period, the higher of task priority.

- EDF is a dynamic scheduling algorithm, and it was proved that EDF is the best dynamical algorithm. The priority of task is dynamically distribute according to the deadline. The shorter of deadline, the higher of task priority.

  RR is designed for time-sharing. In round robin, it define a slice of time to "Time Quantum". Every process has a fixed time to use cpu. When the running time of a process is more than time quantum, the timer will interrupt the process from running state to ready state.

C. *Third milestone*

Overhead in computer science means that the extra move or extra time spending which is irrelevant with target task. So we can compare the overhead time to know the efficacy of the algorithm. Also, we can calculate running time and average response time to compare the strength and weakness of each scheduler.

$$\text{process\_exit time} - \text{first switch time} = \text{running tme} \quad (4)$$

$$\text{switch time} - \text{former switch time} = \text{overhead} \quad (5)$$

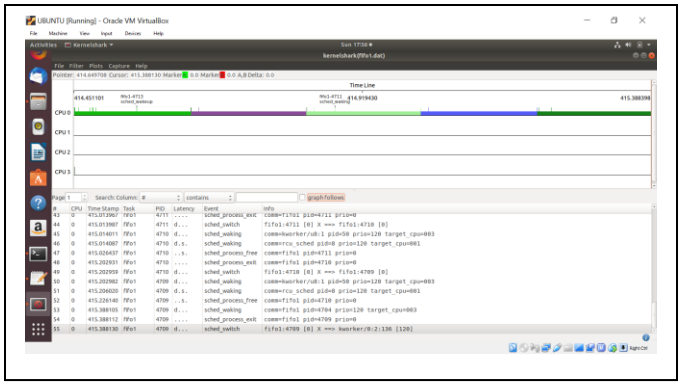
IV. Results

A. *First Milestone*

In the first milestone, we create and trace three different scheduling policy (FIFO, RR and DEADLINE),

1) First case, we constrain five threads to cpu0 which using first-in-first-out policy and the priority is 99. The figure shows that each task has different color, and every task is waking, running and process exit without interrupt.

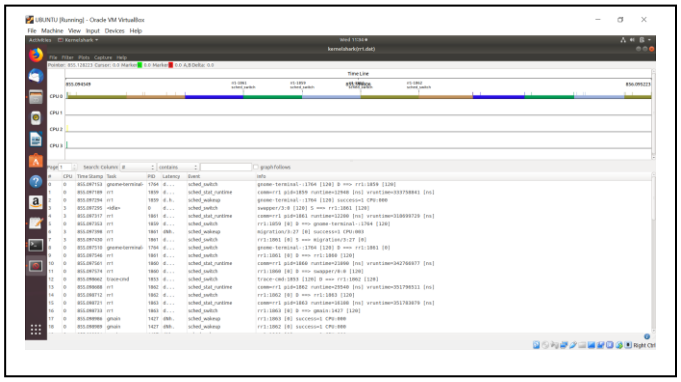
Fig. 1. Simple FIFO scheduling example



```
#   CPU  Time Stamp  Task   PID   Latency  Event               Info
43  0    415.013967  fifo1  4711  ....     sched_process_exit  comm=fifo1 pid=4711 prio=0
44  0    415.013987  fifo1  4711  d...     sched_switch        fifo1:4711 [0] X ==> fifo1:4710 [0]
45  0    415.014011  fifo1  4710  d...     sched_waking        comm=kworker/u8:1 pid=50 prio=120 target_cpu=003
46  0    415.014087  fifo1  4710  d.s.     sched_waking        comm=rcu_sched pid=8 prio=120 target_cpu=001
47  0    415.026437  fifo1  4710  ..s.     sched_process_free  comm=fifo1 pid=4711 prio=0
48  0    415.202931  fifo1  4710  ....     sched_process_exit  comm=fifo1 pid=4710 prio=0
49  0    415.202959  fifo1  4710  d...     sched_switch        fifo1:4710 [0] X ==> fifo1:4709 [0]
50  0    415.202982  fifo1  4709  d...     sched_waking        comm=kworker/u8:1 pid=50 prio=120 target_cpu=003
51  0    415.206020  fifo1  4709  d.s.     sched_waking        comm=rcu_sched pid=8 prio=120 target_cpu=001
52  0    415.226140  fifo1  4709  ..s.     sched_process_free  comm=fifo1 pid=4710 prio=0
53  0    415.388105  fifo1  4709  d...     sched_waking        comm=fifo1 pid=4704 prio=120 target_cpu=003
54  0    415.388112  fifo1  4709  ....     sched_process_exit  comm=fifo1 pid=4709 prio=0
55  0    415.388130  fifo1  4709  d...     sched_switch        fifo1:4709 [0] X ==> kworker/0:2:136 [120]
```

2) Second case, we constrain five threads to cpu0 which using round robin policy and the priority is 99. The figure shows that the tasks are cut into equal slices and running one after another.

Fig. 2. Simple RR scheduling example



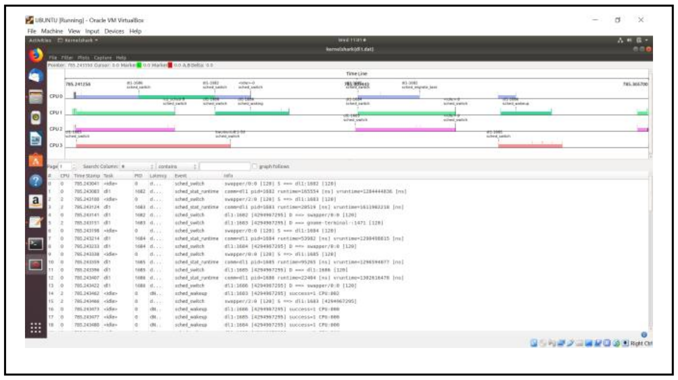
```
#   CPU  Time Stamp  Task            PID   Latency  Event               Info
0   0    855.097153  gnome-terminal- 1764  d...     sched_switch        gnome-terminal-:1764 [120] D ==> rr1:1859 [120]
1   0    855.097189  rr1             1859  d...     sched_stat_runtime  comm=rr1 pid=1859 runtime=12948 [ns] vruntime=333758841 [ns]
2   0    855.097294  rr1             1859  d.h.     sched_wakeup        gnome-terminal-:1764 [120] success=1 CPU:000
3   3    855.097295  <idle>          0     d...     sched_switch        swapper/3:0 [120] S ==> rr1:1861 [120]
4   3    855.097317  rr1             1861  d...     sched_stat_runtime  comm=rr1 pid=1861 runtime=12200 [ns] vruntime=310699729 [ns]
5   0    855.097353  rr1             1859  d...     sched_switch        rr1:1859 [0] D ==> gnome-terminal-:1764 [120]
6   3    855.097398  rr1             1861  dNh.     sched_wakeup        migration/3:27 [0] success=1 CPU:003
7   3    855.097430  rr1             1861  d...     sched_switch        rr1:1861 [0] S ==> migration/3:27 [0]
8   0    855.097510  gnome-terminal- 1764  d...     sched_switch        gnome-terminal-:1764 [120] D ==> rr1:1861 [0]
9   0    855.097546  rr1             1861  d...     sched_switch        rr1:1861 [0] D ==> rr1:1860 [120]
10  0    855.097561  rr1             1860  d...     sched_stat_runtime  comm=rr1 pid=1860 runtime=21000 [ns] vruntime=342766077 [ns]
11  0    855.097574  rr1             1860  d...     sched_switch        rr1:1860 [0] D ==> swapper/0:0 [120]
12  0    855.098662  trace-cmd       1853  d...     sched_switch        trace-cmd:1853 [120] D ==> rr1:1862 [120]
13  0    855.098688  rr1             1862  d...     sched_stat_runtime  comm=rr1 pid=1862 runtime=29540 [ns] vruntime=351798511 [ns]
14  0    855.098712  rr1             1862  d...     sched_switch        rr1:1862 [0] D ==> rr1:1863 [120]
15  0    855.098721  rr1             1863  d...     sched_stat_runtime  comm=rr1 pid=1863 runtime=16100 [ns] vruntime=351783079 [ns]
16  0    855.098733  rr1             1863  d...     sched_switch        rr1:1863 [0] D ==> gmain:1427 [120]
17  0    855.098988  gmain           1427  dNh.     sched_wakeup        rr1:1863 [0] success=1 CPU:000
18  0    855.098989  gmain           1427  dNh.     sched_wakeup        rr1:1862 [0] success=1 CPU:000
```

3) Third case, we create five threads with SCHED_DEADLINE policy. The attribure of each thread can find in table 1. Base on the expample, we can learn that the priority will be the same if the ratio of runtime and deadline is also the same. Another face is that the time interval between the same task is base on deadline, and thread running time is base on runtime.

TABLE I. Threads Attribute in Third Case

Task Color	***Thread attribute (unit: nano-second)***		
	runtime	***period***	***deadline***
Blue	10000000	30000000	20000000
pink	20000000	60000000	40000000
Light green	20000000	60000000	40000000
Orange	30000000	90000000	60000000
Dark green	30000000	90000000	60000000

Fig. 3. Simple DEADLINE scheduling example



```
#   CPU  Time Stamp  Task    PID   Latency  Event               Info
0   0    785.243041  <idle>  0     d...     sched_switch        swapper/0:0 [120] S ==> dl1:1682 [120]
1   0    785.243083  dl1     1682  d...     sched_stat_runtime  comm=dl1 pid=1682 runtime=165554 [ns] vruntime=1284444836 [ns]
2   2    785.243100  <idle>  0     d...     sched_switch        swapper/2:0 [120] S ==> dl1:1683 [120]
3   2    785.243124  dl1     1683  d...     sched_stat_runtime  comm=dl1 pid=1683 runtime=28519 [ns] vruntime=1611982218 [ns]
4   0    785.243141  dl1     1682  d...     sched_switch        dl1:1682 [4294967295] D ==> swapper/0:0 [120]
5   2    785.243151  dl1     1683  d...     sched_switch        dl1:1683 [4294967295] D ==> gnome-terminal-:1471 [120]
6   0    785.243198  <idle>  0     d...     sched_switch        swapper/0:0 [120] S ==> dl1:1684 [120]
7   0    785.243214  dl1     1684  d...     sched_stat_runtime  comm=dl1 pid=1684 runtime=53982 [ns] vruntime=1298498815 [ns]
8   0    785.243233  dl1     1684  d...     sched_switch        dl1:1684 [4294967295] D ==> swapper/0:0 [120]
9   0    785.243338  <idle>  0     d...     sched_switch        swapper/0:0 [120] S ==> dl1:1685 [120]
10  0    785.243359  dl1     1685  d...     sched_stat_runtime  comm=dl1 pid=1685 runtime=95265 [ns] vruntime=1296594077 [ns]
11  0    785.243398  dl1     1685  d...     sched_switch        dl1:1685 [4294967295] D ==> dl1:1686 [120]
12  0    785.243407  dl1     1686  d...     sched_stat_runtime  comm=dl1 pid=1686 runtime=22404 [ns] vruntime=1302618478 [ns]
13  0    785.243422  dl1     1686  d...     sched_switch        dl1:1686 [4294967295] D ==> swapper/0:0 [120]
14  2    785.243462  <idle>  0     dN..     sched_wakeup        dl1:1683 [4294967295] success=1 CPU:002
15  2    785.243466  <idle>  0     d...     sched_switch        swapper/2:0 [120] S ==> dl1:1683 [4294967295]
16  0    785.243473  <idle>  0     dN..     sched_wakeup        dl1:1686 [4294967295] success=1 CPU:000
17  0    785.243477  <idle>  0     dN..     sched_wakeup        dl1:1685 [4294967295] success=1 CPU:000
18  0    785.243480  <idle>  0     dN..     sched_wakeup        dl1:1684 [4294967295] success=1 CPU:000
```

B. *Second Milestone*

In the second milestone, we create 6 task to compare RM with SCHED_FIFO, EDF with SCHED_DEADLINE, and RR with SCHED_RR.

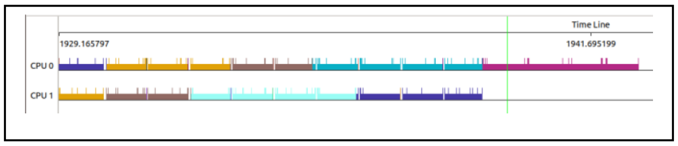
The setting of each scheduling algorithm is in the table and the result of kernelshark shows in figure.

1) In rate monotonic algorithm, the first three tasks are yellow, brown and dark blue. Base on the event order, the first two task are yellow and dark blue. After a tine quantum, the yellow task is moved to cpu0 and dark blue task is replaced by brown because of brown task has higher priority. At three seconds, the light blue task has higher priorty than brown task. After yellow task finished, the brown and blue tasks are running after. Finally, the lowest priority tasks (dark blue, dark pink) run last.

TABLE II. RATE MONOTONIC CASE

First Task	***Second Task***	***Schedule attribute***	
		Priority	***Period***
Yellow	Light Blue	90	3s
Brown	Blue	80	4s
Dark Blue	Dark Pink	70	5s

Fig. 4. Rate monotonic case scheduling example



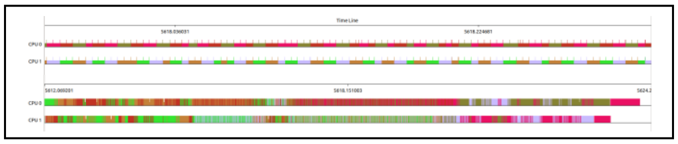
2) In earlier deadline first algorithm, we can see that the tasks are cut into very small pieces and interweave to each other. As show in figure 5, the upper figure is the zoom in example of buttom figure.

The first three tasks are green, red and brown. But you can notice that the green task is more than red task, and the read task is more than brown task according to the deadline order. At three seconds the purple task start running and the dray task running one second later. After the first two tasks finished, other tasks running like FIFO and finished.

TABLE III. RATE MONOTONIC CASE

First Task	***Second Task***	***Schedule attribute***	
		Priority	***Period***
Light Brown	Dark Orange	90	3s
Pink	Green	80	4s
White	Dark Pink	70	5s

Fig. 5. Earlier deadline first case scheduling example



3) In round robin algorithm, the first three tasks are lightbrown pink and white. According to the executing order, pink and white occupy first slice. After a time quantum, the light brown and pink are running because of their high priority. At three seconds, dark orange execute in cpu1. At four seconds, green execute in cpu0. At five seconds, dark pink start executing but did not in any cpu ecause of the priority is lower than dark orange and green. After dark orange finished, the white and pink start together with changing task slices. When green finished, white and dark pink are running in cpu0 and cpu1.

Fig. 6. Rate monotonic case scheduling example

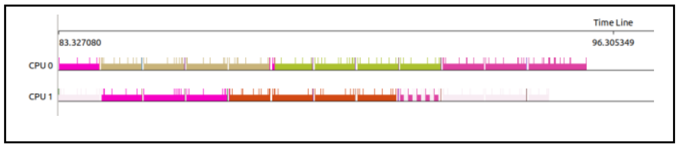


TABLE IV. RATE MONOTONIC CASE

First Task	***Second Task***	***Schedule attribute (unit: nano-second)***		
		Runtime	***Period***	***Deadline***
Green	Purple	10000000	30000000	30000000
Red	Gray	10000000	40000000	40000000
Brown	Pink	10000000	50000000	50000000

C. Third Milestone

1) According to the data submission, the runtime of RM is 13.64065 sec in cpu0, 9.972036 in cpu1. Total runtime is 23.61269 sec. The runtime of RR is 12.33396 sec in cpu0, 10.473448 sec in cpu1. Total runtime is 22.807408.

2) The total overhead of RM is 7.72231 sec which is lesser than RR (9.938956 sec). The response time of RR is 8.4414985 sec which is more than RM (8.426626633 sec).