OS Project2 Report

Ning Li

May 18, 2022

1 Introduction

Tracing memory access for tasks(processes) is useful in Android systems. We can use these traces to compute the race probabilities between different tasks. Further more, we can use the race probabilities for better scheduling. In Linux, although the hardware uses a R bit and a M bit to track if a page has been referenced or modified, it does not maintain access frequency for each task. Thus, in the first part of this project, we tried to track the access frequency in software.

CPU scheduling is the basis of multiprogrammed operating systems. When the Linux kernel scheduler chooses a task, it considers a variety of factors. However, it doesn't consider the likelihood that a task may race with one of the currently running tasks. In the second part of this project, based on round robin scheduling algorithm, we implemented a race-averse scheduling algorithm(RAS) to the Linux kernel, to let the kernel schedule a task that is least likely to race with any of the currently running tasks.

In this project, we did some extra work. Firstly, we try to build RAS in a multi-cpu architecture and implement load balance. Then, We compared the performance of the RAS scheduler with the existing schedulers, to analysis their pros and cons. Besides, we improved the RAS scheduler, making it recognize the foreground tasks and background tasks and allocate different timeslices for these two kind of tasks.

2 Page Access Tracing

2.1 Analysis and Implementation

Firstly, in the `task_struct`, we add a integer variable `wcounts` to record the write times and a boolean variable `trace_flag` to determine when to start trace. In `include/linux/init_task.h`, we add `wcounts` and `trace_flag` in `INIT_TASK` for initialization.

In order to support page access tracing of a task, We use `mprotect` to protect specific memory space. If the task try to write data to the memory space, there will be a page write fault. Then function `do_page_fault` will handle the problem. It will call another function `__do_page_fault` to judge the error type. `__do_page_fault` calls `access_error` to ensure if the page has write permission. If not, it will return corresponding error type. Therefore, if the `trace_flag` is true, we can increase `wcounts` before it returns.

We define three system call to provide the interface of page access tracing.

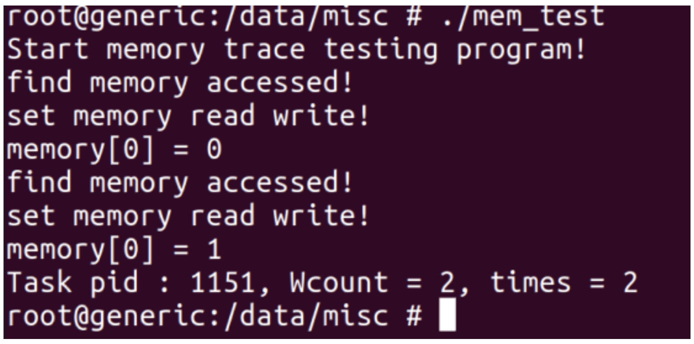
- `start_trace`
  This system call sets `trace_flag` to be true, and sets `wcounts` to zero. If `trace_flag` is already true, which means the task has been tracing, the system call will return -EINVAL.
- `stop_trace`
  This system call sets `trace_flag` to be false.

- `get_trace`
  This system call accesses task's `wcounts` and put it in the given integer address.

2.2 Testing

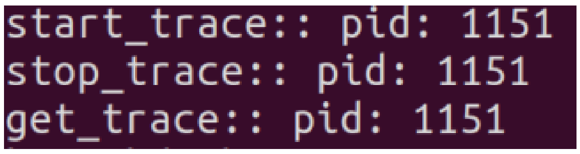
We use the process `mem_test` to test the page access tracing mechanism. The process starts page sccess tracing and allocates a set of memory, which can only be read. After that it try to write data to this memory space, and will receive a SIGSEGV. Then the function `__do_page_fault` will increase `wcounts`. At the same time, `segv_handler` will handle SIGSEGV by giving the memory write permission. After writing to this memory successfully, the process use `mprotect` to protect the memory again. If it try to write data to this memory space again, the same thing mentioned above will happen again. Finally, the process output the wcounts to show the result. Figure 1 and Figure 2 show the test result.

Here, we test page access tracing for only one process. We will test multiprocesses in the test of next part.



```
root@generic:/data/misc # ./mem_test
Start memory trace testing program!
find memory accessed!
set memory read write!
memory[0] = 0
find memory accessed!
set memory read write!
memory[0] = 1
Task pid : 1151, Wcount = 2, times = 2
root@generic:/data/misc #
```

Figure 1: `mem_test`: adb shell result



```
start_trace:: pid: 1151
stop_trace:: pid: 1151
get_trace:: pid: 1151
```

Figure 2: `mem_test`: kernel result

3 Race-Averse Scheduler

3.1 Analysis and Implementation

3.1.1 `ras.c`, the scheduler

Linux uses the `sched_class` structure to implement the scheduler, which abstracts different scheduling mechanism. Our scheduler implementation is mainly to implement the `sched_class` structure of the RAS scheduler in `/kernel/sched/ras.c`.

RAS is a weighted round robin style scheduling according to race probabilities of each tasks. We assume that a large enough number of tasks are in scheduling, and they may compete at a given range of virtual addresses. Here we show how to calculate the race probability, the weight and the timeslice.

The race probability of a task is represented as an integer in the range of [0, 10]. The higher the integer is, the more likely that the tasks may race with others. The race probability of a certain task is proportional to its frequency of accessing the given range of virtual addresses. Equation 1 shows how to calculate the race probility , which avoids floating number division and conversion between integer and floating number.

$$race\ probility = \begin{cases} 0, \quad if\ wcounts = 0 \text{ (avoid the case that } total\ wcounts = 0) \\ \frac{wcounts}{\lceil total\ wcounts\ /\ 10 \rceil}, \quad Others \end{cases} \tag{1}$$

The weight of a task is represented as an integer in the range of (0, 10]. Equation 2 shows how to calculate the weight.

$$weight = \begin{cases} 1, \quad if\ race\ probability = 10 \\ 10 - race\ probability, \quad Others \end{cases} \tag{2}$$

At last we calculate the timeslice, shown in equation 3. $RAS\_TIMESLICE = 10ms$, so the total timeslice is in the range of $10ms$ to $100ms$.

$$timeslice = weight * RAS\_TIMESLICE \tag{3}$$

The number of tasks on a RAS run queue are not constant, which means that total wcounts is changing. Besides, the wcounts of a task is also changing. So we need to update the timeslice at the appropriate time. Initially, when a task enter the RAS run queue, we will update it's timeslice(the function `enqueue_task_ras` calls `update_time_slice_ras`). Besides, when a running task use up its timeslice and will be put in the tail of the RAS run queue, we update its timeslice(the function `task_tick_ras` calls `update_time_slice_ras` in the end). It seems better if we update every task's timeslice, because the change of total wcounts will affact every task's timeslice. However, we can't accept the time complexity of O(n).

The major part of RAS is in `kernel/sched/ras.c` , we mainly implement these following functions:

- `init_ras_rq` : Initialize the ras run queue.
- `update_curr_ras` : Update the current task's runtime statistics. Skip current tasks that are not in our scheduling class.
- `update_time_slice_ras` : Update the time_slice of a task.
- `enqueue_task_ras` : Adding a task to a ras run list.
- `dequeue_task_ras` : Remove a task from ready queue of RAS.
- `requeue_task_ras` : Put task to the head or the end of the run list without the overhead of dequeue followed by enqueue.
- `yield_task_ras` : Move task to the tail of ready queue.
- `pick_next_task_ras` : Pick the next task to run.If there is no ready task or the current round of RAS has reached the time limit, it returns a NULL pointer.
- `put_prev_task_ras` : Put the previous task back to the tail of ready queue if needed.
- `set_curr_task_ras` : Set the start time of execution.
- `task_tick_ras` : This function is invoked every 1ms and will update the remaining time slice of the task. When it finds that the task time slice is exhausted, it will recalculate the time slice and mark the task as needing rescheduling.

- `get_rr_interval_ras` : Return the timeslice of a task.
- `switched_to_ras` : Do some work after switching to the `sched_ras` class.

3.1.2 Adding RAS scheduler to kernel

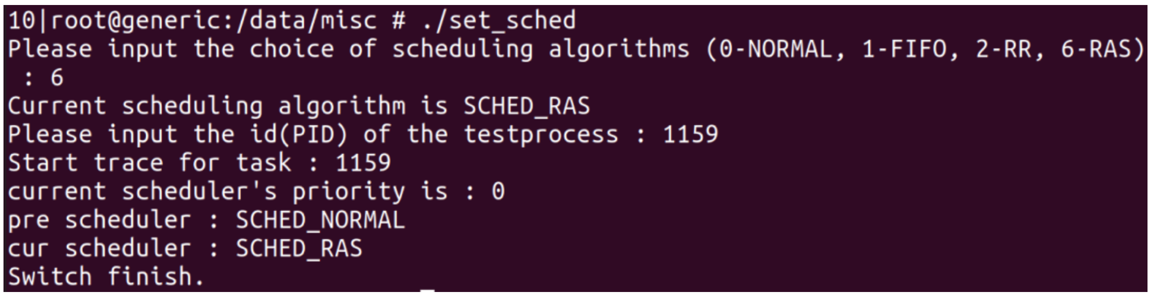
In order to add our RAS scheduler to the kernel, we need to modify some existing files in the Android kernel:

- `/arch/arm/configs/goldfish_armv7_defconfig`
  Add `CONFIG_RAS_GROUP_SCHED` option and set it to y.
- `/include/linux/sched.h`
  Define `SCHED_RAS` with value 6 to indicate RAS policy.
  Define `sched_ras_entity` structure to store necessary information. It includes: `run_list`: a `link_head` instance to link neighbor tasks; `time_slices`: indicate the time for every RR turn; `weight`: indicate the weight of a task; `old_wcounts`: indicate the old wcounts. When updating a task's timeslice, total wcounts will subtract its old wcounts and plus its current wcounts.
  Define `RAS_TIMESLICE` with const value $10ms$.
  Add a `sched_ras_entity` varaible to `task_struct`.
  Declare a `ras_rq` struct.
- `include/linux/init_task.h`
  Add a ras member in `INIT_TASK` for initialization.
- `/kernel/sched/sched.h`
  Define structure `ras_rq` , the ready queue of RAS. In `ras_rq`, `run_list` is a `link_head` instance to link tasks. `ras_nr_running` records the number of tasks on the ready queue. `total_wcounts` records the total wcounts of all tasks on this queue.
  Add a member `ras` of type `ras_rq` to structure rq.
  Declare some extern variables and functions.
- `/kernel/sched/core.c`
  Revise the function `rt_mutex_setprio`, `__setscheduler`, `__sched_setscheduler`, `__sched_fork`, `for_each_possible_cpu` to make RAS works well.
- `kernel/sched/fair.c`
  Modify `.next` in `fair_sched_class` to make it point to `ras_sched_class`.
- `kernel/sched/Makefile`
  Add `ras.o` for compilation.

3.2 Testing

3.2.1 Scheduling Policy Conversion

We use the function `sched_setscheduler` to change task's scheduling policy and use the function `sched_setscheduler` to access the previous and current scheduling policy. Figure 3 shows the test result.



```
10|root@generic:/data/misc # ./set_sched
Please input the choice of scheduling algorithms (0-NORMAL, 1-FIFO, 2-RR, 6-RAS)
 : 6
Current scheduling algorithm is SCHED_RAS
Please input the id(PID) of the testprocess : 1159
Start trace for task : 1159
current scheduler's priority is : 0
pre scheduler : SCHED_NORMAL
cur scheduler : SCHED_RAS
Switch finish.
```

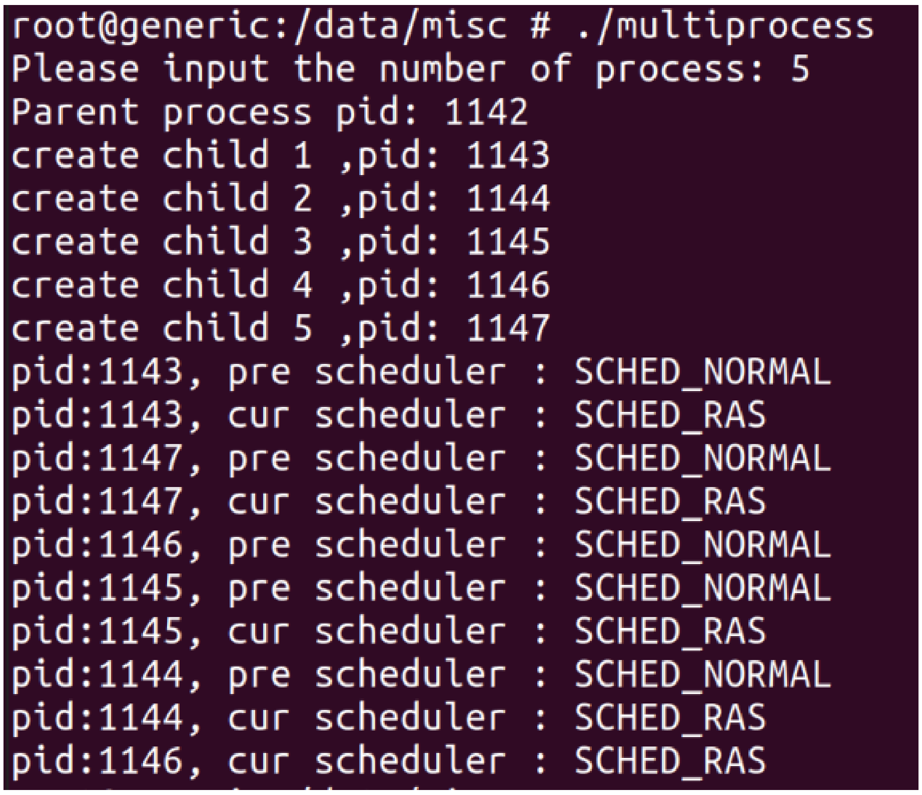
Figure 3: `set_sched`: adb shell result

3.2.2 Multiprocesses

Here we create several processes and change their `wcounts` by doing same operations mentioned in section 2.2. Figure 4 and Figure 5 show the test result.

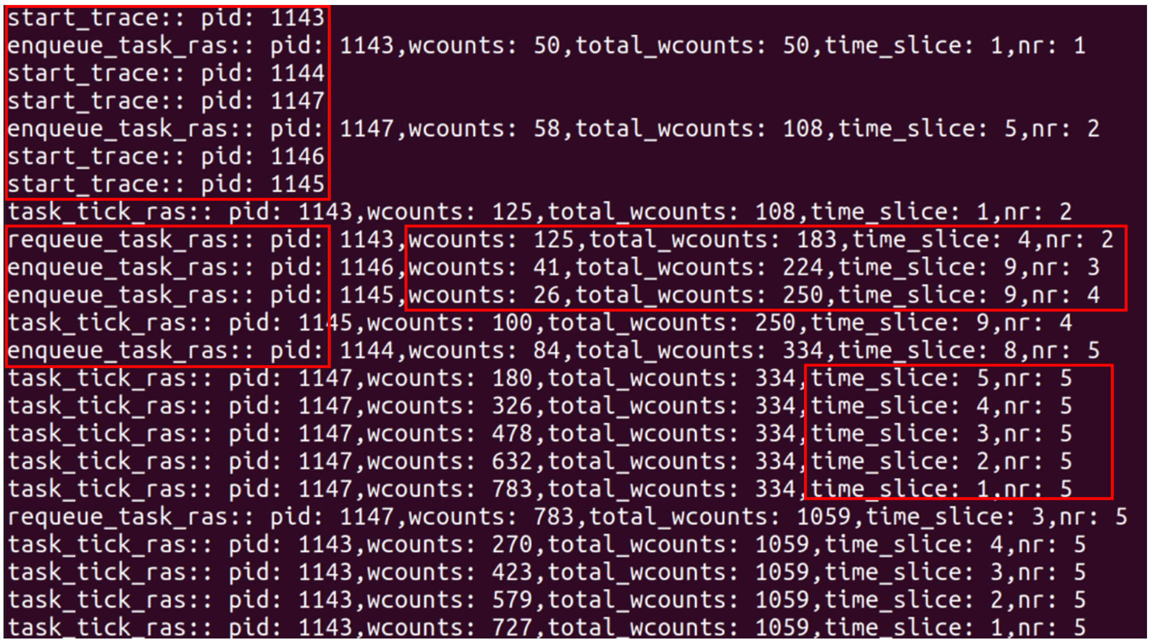
From Figure 4 we can see that all of the five processes change their scheduling policy to RAS successfully.

From Figure 5 we can see that the page access tracing mechanism works well in multiprocesses. Besides, processes can enqueue, requeue and use their timeslice successfully with RAS. We can also see that with different `wcounts`, the tasks will get different timeslices.



```
root@generic:/data/misc # ./multiprocess
Please input the number of process: 5
Parent process pid: 1142
create child 1 ,pid: 1143
create child 2 ,pid: 1144
create child 3 ,pid: 1145
create child 4 ,pid: 1146
create child 5 ,pid: 1147
pid:1143, pre scheduler : SCHED_NORMAL
pid:1143, cur scheduler : SCHED_RAS
pid:1147, pre scheduler : SCHED_NORMAL
pid:1147, cur scheduler : SCHED_RAS
pid:1146, pre scheduler : SCHED_NORMAL
pid:1145, pre scheduler : SCHED_NORMAL
pid:1145, cur scheduler : SCHED_RAS
pid:1144, pre scheduler : SCHED_NORMAL
pid:1144, cur scheduler : SCHED_RAS
pid:1146, cur scheduler : SCHED_RAS
```

Figure 4: `multiprocess`: adb shell result



```
start_trace:: pid: 1143
enqueue_task_ras:: pid: 1143,wcounts: 50,total_wcounts: 50,time_slice: 1,nr: 1
start_trace:: pid: 1144
start_trace:: pid: 1147
enqueue_task_ras:: pid: 1147,wcounts: 58,total_wcounts: 108,time_slice: 5,nr: 2
start_trace:: pid: 1146
start_trace:: pid: 1145
task_tick_ras:: pid: 1143,wcounts: 125,total_wcounts: 108,time_slice: 1,nr: 2
requeue_task_ras:: pid: 1143,wcounts: 125,total_wcounts: 183,time_slice: 4,nr: 2
enqueue_task_ras:: pid: 1146,wcounts: 41,total_wcounts: 224,time_slice: 9,nr: 3
enqueue_task_ras:: pid: 1145,wcounts: 26,total_wcounts: 250,time_slice: 9,nr: 4
task_tick_ras:: pid: 1145,wcounts: 100,total_wcounts: 250,time_slice: 9,nr: 4
enqueue_task_ras:: pid: 1144,wcounts: 84,total_wcounts: 334,time_slice: 8,nr: 5
task_tick_ras:: pid: 1147,wcounts: 180,total_wcounts: 334,time_slice: 5,nr: 5
task_tick_ras:: pid: 1147,wcounts: 326,total_wcounts: 334,time_slice: 4,nr: 5
task_tick_ras:: pid: 1147,wcounts: 478,total_wcounts: 334,time_slice: 3,nr: 5
task_tick_ras:: pid: 1147,wcounts: 632,total_wcounts: 334,time_slice: 2,nr: 5
task_tick_ras:: pid: 1147,wcounts: 783,total_wcounts: 334,time_slice: 1,nr: 5
requeue_task_ras:: pid: 1147,wcounts: 783,total_wcounts: 1059,time_slice: 3,nr: 5
task_tick_ras:: pid: 1143,wcounts: 270,total_wcounts: 1059,time_slice: 4,nr: 5
task_tick_ras:: pid: 1143,wcounts: 423,total_wcounts: 1059,time_slice: 3,nr: 5
task_tick_ras:: pid: 1143,wcounts: 579,total_wcounts: 1059,time_slice: 2,nr: 5
task_tick_ras:: pid: 1143,wcounts: 727,total_wcounts: 1059,time_slice: 1,nr: 5
```

Figure 5: `multiprocess`: kernel result

4 Additional Work

4.1 Multi-CPU Architecture

In this section, we try to build RAS in a multi-cpu architecture and implement load balance. For simplicity, we mainly implement the function `select_task_rq_ras` in `ras.c` to support multi-cpu architecture. Since the total wcounts is the number of memory accessing of all tasks on a RAS run queue, we use the total wcounts to roughly reflect the load of a CPU. When a new task is going to execute, we select the available CPU with the smallest total wcounts. In this way, it guarantees the load balance among CPUs.

Unfortunately, the provided emulator is a uniprocessor archecture. We can't test the performance of multi-cpu architecture load balance with RAS.

4.2 Scheduler Comparison

In Linux, there are mainly three scheduler: NORMAL(CFS), FIFO and RR. Now we want to conpare the performance of RAS and the three original schedulers. We create different number of tasks, change their scheduling policies and make them do different number of memory accessing. We record every task's runtime and analyze their differences and similarities.

First, we fork 10 tasks. Every task does memory accessing for random times in the range of 1 to 1000. Figure 6 shows the result. The runtime of RAS, RR(priority 99) and FIFO(priority 99) is very close. The reslut is expected, because the runtime is less than 100ms. The timeslice of RR and the initial timeslice are both 100ms, so every task will finish executing in one scheduling cycle. The runtime of NORMAL is longest, which is 262ms averagely.

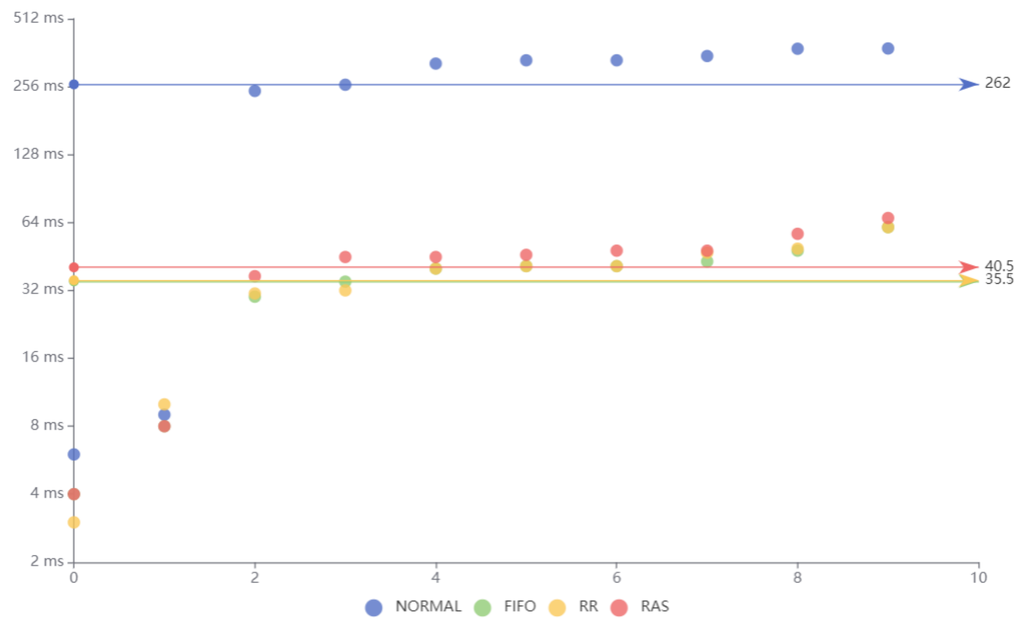


Figure 6: Runtime of 10 tasks: (0,1000] times memory accessing

Then, we fork 50 tasks. Every task also does memory accessing for random times in the range of 1 to 1000. Figure 7 shows the result. The result is similar to 10 tasks. a small difference is RAS's runtime is closer to RR and FIFO's runtime. It corresponds with the assumption that a large enough number of tasks are in scheduling.

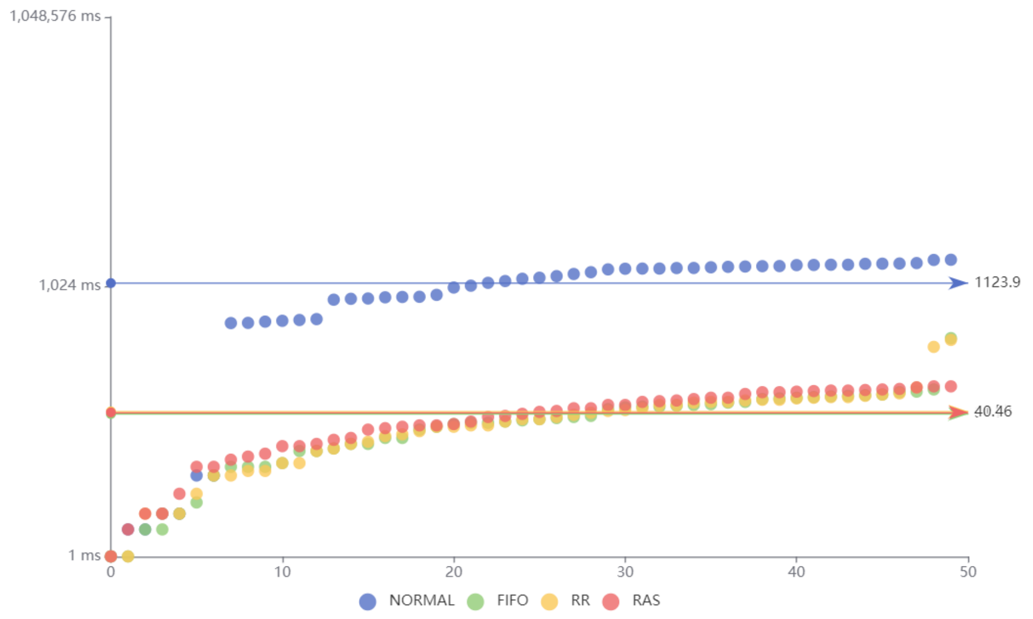


Figure 7: Runtime of 50 tasks: (0,1000] times memory accessing

Finally, we also fork 50 tasks. However, we increase the times of memory accessing, which is in the range of 1 to 5000 now. Figure 8 shows the result. We can see that the runtime of FIFO is best, far less than others. The runtime of RR and RAS is very close. Similarly, the performance of NORMAL is the worst.

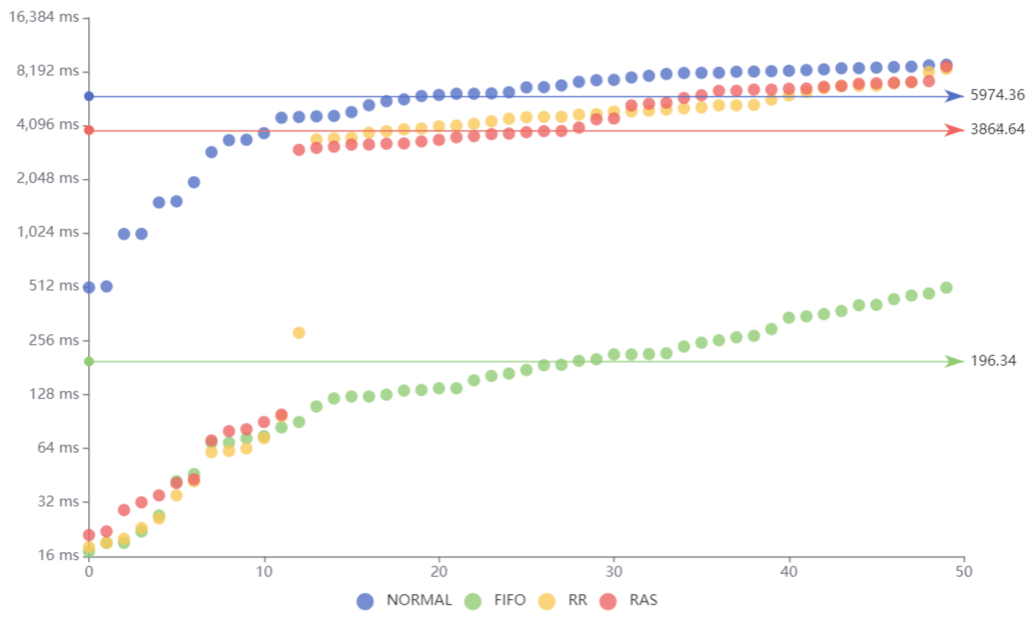


Figure 8: Runtime of 50 tasks: (0,5000] times memory accessing

From these conparisons we can make a conclusion that the RAS scheduler is most similar to the RR scheduler. Their performances are almost same with a large number of tasks. After all, RAS is a round robin based scheduler. Besides, it seems that the FIFO scheduler is the best, but sometimes it will result in starvation. Every scheduler has its pros and cons.

4.3 Foreground and background task

Android tasks can be classified into foreground groups and background groups. We want to assign different milliseconds as a timeslice for background groups and foreground groups. We give the background task less timeslice. In this way, the foreground will obtain more CPU time relatively. And it can finish its job more quickly.

In the file `/include/linux/sched.h`, we use the existing `RAS_TIMESLICE` as the basic timeslice for foreground tasks. For background tasks, we define `RAS_BG_TIMESLICE` with const value $5ms$. When we calculate the timeslice in function `update_time_slice_ras` defined in `ras.c`, we first recognize whether it is a foreground task or background task. We use the function `task_group_path` from `kernel/ sched/debug.c` to get task's group path and find out whether it is a foreground task or a background task. If it's a background task, we will change the equation 3 into the equation 4.

$$timeslice = weight * RAS\_BG\_TIMESLICE \tag{4}$$

5 Conclusion

From this project, I learned more about the scheduling algorithms in Linux. When implementing the RAS scheduler, from simple imitation to adding my own ideas, I felt a sense of achievement. I believe this project will be beneficial for my future programming practice and further learning of operating system.

At last, I'd like to express my sincere thanks for Prof. Wu, Prof. Chen, TA Huo, TA Zeng and all my classmates.