# Project 2 for Operating System
# **Android Race-Averse Scheduler**

Weiming Zhang

May 26, 2022

## 1 Introduction

A process is an instance of a computer program that is being executed. Each process has its own address space. A thread is also the programmed code in execution, which is the smallest unit that can be managed independently by a kernel scheduler. In modern computer systems, there are many threads waiting to be served at the same time. So the job of the scheduler in the kernel is to decide which thread to run for how long.

Linux systems developed different kinds of scheduling algorithms for different situations of scheduling tasks. For instance, the CFS Scheduler for normal processes and the RT Scheduler for real-time processes. They schedule different tasks on the basis of priorities from their perspective.

In this project, we implement a basic Race-Averse Scheduler in a weighted round robin style and add it to the Linux kernel. We associate the tasks with its tendency to access the virtual memory as well as the likelihood that it may race with one of the currently running tasks. We define race probabilities to decide how many timeslices the scheduler provides. To achieve this, we need to realize per-task page access tracing, which can trace the number of times a page is written by a particular task.

## 2 Basic Analysis

### 2.1 Overview of Process Scheduling

The operating system manage large amounts of processes and it has different scheduling policies to decide which process to run. To achieve this effectively, there is a running queue, in which the scheduler chooses the next task.

There already exist some scheduling policies in the Linux Kernel. The policies cater for different categories of processes: SCHED_FIFO, SCHED_RR for real-time processes, and SCHED_NORMAL, SCHED_IDLE for normal processes.

- SCHED_FIFO: First-in-first-out policy: it can run all along until preempted
- SCHED_RR: round-robin policy: run until its timeslices are used up
- SCHED_NORMAL: completely-fair-scheduling policy: most of the kernel processes use this policy
- SCHED_IDLE: for leisure situations

## 2.2 Scheduling Class

Corresponding to different scheduling policies, there are different scheduling classes. The different scheduling classes are arranged in a fixed order, representing their relative priorities. The schedule function in the core.c would traverse the set of scheduling classes and choose the due class to choose the next process in its running queue. For example, real-time processes enjoy scheduling classes with higher priorities, which will be more likely to chosen compared with fair_sched_class.

Then there are some general functions that a scheduling class should contain:

```
struct sched_class {
    const struct sched_class *next;

    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
    void (*yield_task)   (struct rq *rq);

    void (*check_preempt_curr)(struct rq *rq, struct task_struct *p, int flags)
        ;

    struct task_struct * (*pick_next_task)(struct rq *rq,
                              struct task_struct *prev,
                              struct rq_flags *rf);
    void (*put_prev_task)(struct rq *rq, struct task_struct *p);

    void (*set_curr_task)(struct rq *rq);
    void (*task_tick)(struct rq *rq, struct task_struct *p, int queued);

    void (*update_curr)(struct rq *rq);
};
```

- next: the pointer, to compose the list of scheduling classes
- enqueue_task: to put a runnable process into the running queue
- dequeue_task: to remove a current process from the running queue
- yield_task: the current process gives up the cpu resources
- check_preempt_curr: call a newly-waked-up process to preempt the current process
- pick_next_task: choose the next process to execute
- put_prev_task: replace the current executing process
- set_curr_task: call it if the scheduling policy is changed
- task_tick: check if the current thread is running for too long and whether need replaced
- update_curr: update the information of the running queue

Each scheduling class should be implemented around the functions above, including the RAS policy we implement. In this project, we place ras_sched_class between the fair_sched_class and the idle_sched_class to run normally.

### 2.3 Core Scheduler

We've shown a basic framework of scheduling work above. But how it is put into work is decided by the core scheduler. It is a huge job, which is mainly defined in the /kernel/sched/core.c.

The __schedule() is the main body of the core scheduler. What it does includes putting the previously running thread into a run queue, picking a new thread to run next, and lastly switching context between the two threads. Most work in __schedule() is delegated to the scheduling classes. If a function is invoked, then actual work is done by the function registered to the function pointer of the schduling class that the previously running task belongs to.

Another important function is the scheduler_tick(), which is the heart of the scheduler. The tick function is periodically called by the kernel with frequency HZ defined in the /include/linux/sched.h, which is the tick rate of the system timer defined on system boot. The tick invokes update_-rq_clock(), reading a clock source and updates the clock of the run queue and check whether a replacement is needed.

## 3 Implementation Details

### 3.1 Page Trace

To realize the trace of the memory access, we define some new variables in the task_struct. One is wcounts to count the write operation of a task according to the page fault handler, which will send a fault signal to the process if it wants to write into protected page. Another variable is trace_flag, representing whether the task is tracing.

To do this, at first we modify task_struct in /include/linux/sched.h:

```
struct task_struct{
    volatile long state;
    ...
    unsigned int wcounts;
    unsigned int  trace_flag;
}
```

And then we need to modify the page fault strategy, to do some changes to the newly added variables in the task_struct when page fault happens. So corresponding modifications are needed in the /arch/arm/mm/fault.c.

In our project, we only need to cope with page write fault, so we need to modify the do_page_-fault function. But the core of the problem is to deal with the access error, because all satisfying page write faults need to go through it. So we directly do some changes in the access_error function.

```
static inline bool access_error(unsigned int fsr, struct vm_area_struct *vma)
{
    unsigned int mask = VM_READ | VM_WRITE | VM_EXEC;
    struct task_struct *tsk;
    tsk = current;
    if (fsr & FSR_WRITE)
    {
        mask = VM_WRITE;
        if (tsk->trace_flag && !(vma->vm_flags & VM_WRITE))
        {
            tsk->wcounts++;
            printk(KERN_INFO "WRITE FAULT!");
```

```
                printk(KERN_INFO "wcounts=%d", tsk->wcounts);
            }
        }
        ...
}
```

So far we've completed our modifications to the kernel files. We also need to implement some system calls to put our job in effect. We need three syscalls, to start, get and stop tracing respectively.

Here we list their functions to realize, only including some trivial changes to the wcounts and trace_flag.

- start_trace.c: Set the wcounts to 0 and set the trace_flag to 1
- stop_trace.c: Set the wcounts to 0 and set the trace_flag to 0
- get_trace.c: Return the wcounts

## 3.2 Define RAS structures

In the previous subsection we've completed the page trace mechanism. Here we need to insert some structures of our RAS scheduler into the scheduling-related files.

Because the Race-Averse Scheduling policy is based on the weighted-round-robin scheduling policy, so we can refer to the real-time scheduling including FIFO and RR and design our RAS based on them. Most of the implementations are similar to that in RR scheduler.

At first, we need to revise the /include/linux/sched.h. We can find that there already exist some scheduling policies defined here. So we define a SCHED_RAS to be 6.

The task_struct is a per process struct used to store the information of the process. In scheduling, we need further information to manage the possible process executions, so scheduling entities are needed and implemented in this file. We can regard it as a node in the corresponding queue. Similar to the sched_rt_entity owned by real-time processes, we define sched_ras_entity for our RAS scheduling.

```
struct sched_ras_entity{
    struct list_head run_list;//insert into the specific priority queue
    unsigned long timeout;
    unsigned int time_slice;//the timeslice left for the current  process
    unsigned int race_prob;//the race probability of the current process

    struct sched_ras_entity *back;
    struct sched_ras_entity *parent;
    struct ras_rq *ras_rq;
    struct ras_rq *my_q;
};
```

And we need to do more things to accomplish the job in this file:

- add a sched_ras_entity into the task_struct
- declare a ras_rq struct next to struct cfs_rq and rt_rq

Because our basic implementation does not need some declarations about the priorities, we just move on and see something about the ras_rq.

The ras_rq struct along with the rt_rq struct is defined in the /kernel/sched/sched.h, where most of the scheduling information exists.

There is a rq struct defined in it. That is the running queue seen by the core scheduler, and in the struct there are sub running queues. Choose runnable processes in different scheduling classes, and they will be placed in the corresponding sub running queue. And it's not hard to understand that's where the entities flow.

Our basic ras_rq is much simpler than the rt_rq, we can just take it as a common list.

```
struct ras_rq{
    struct list_head queue;
    struct ras_prio_array active;
    unsigned long ras_nr_running;//the number of running processes
    raw_spinlock_t ras_runtime_lock;

    struct rq *rq;//running queue
    struct list_head leaf_ras_rq_list;
};
```

And we should add the ras_rq into the rq struct, just like the rt_rq.

Then we need to do some modifications in the core scheduler in kernel/sched/core.c.

- ___sched_fork: add INIT_LIST_HEAD(p->ras.run_list)
- ___set_scheduler: add a conditional branch: if(p->policy==SCHED_RAS)p->sched_class=ras_-sched_class
- ___set_scheduler: INVAL returning: if(policy!=SCHED_RAS)
- sched_init: add function init_ras_rq(rq->ras,rq)

More work needs to be done in our implementation of RAS scheduler.

### 3.3 RAS Schduling Class Implementation

We should implement the necessary functions in sched_class to make our RAS Scheduler work normally. Most of the functions are similar to the corresponding functions in rt.c, so we just do some revisions and show some of the details here.

- enqueue_task_ras: We enqueue a new process into the running queue by adding it to the runlist of the running entity by list_add. We also set an initial timeslice for a newly enqueued process.

```
static void enqueue_ras_entity(struct rq *rq, struct sched_ras_entity *ras_se,
                               bool head) {
  struct list_head *queue = &(rq->ras.queue);
  struct task_struct *p;
  p = container_of(ras_se, struct task_struct, ras);
  p->ras.time_slice = 10;
  if (head)
    list_add(&ras_se->run_list, queue);
  else
    list_add_tail(&ras_se->run_list, queue);
  ++rq->ras.ras_nr_running;
}

static void enqueue_task_ras(struct rq *rq, struct task_struct *p, int flags) {
  printk("Enqueue An New RAS Task!\n");
  struct sched_ras_entity *ras_se = &p->ras;
  enqueue_ras_entity(rq, ras_se, flags & ENQUEUE_HEAD);
  inc_nr_running(rq);
}
```

As we can see, we first get some pointers according to the parameters and use them in the detailed but simple functions.

- dequeue_task_ras: Opposite to the enqueue function above, here we try to dequeue an executed process. We remove one from the current scheduling list, and decline the nr_-running.

```
static void dequeue_ras_entity(struct rq *rq, struct sched_ras_entity *ras_se)
    {
  list_del_init(&ras_se->run_list);
  rq->ras.ras_nr_running--;
}

static void dequeue_task_ras(struct rq *rq, struct task_struct *p, int flags) {
  printk("Dequeue An RAS Task!\n");
  struct sched_ras_entity *ras_se = &p->ras;
  update_curr_ras(rq);
  dequeue_ras_entity(rq, ras_se);
  dec_nr_running(rq);
}
```

- pick_next_task: The RAS Scheduler try to look for the next process to be executed. Because we do not consider the effects of priorities, we just choose the first one in the run_list. In fact, we just simplify a little bit to the rt scheduler.

```
static struct task_struct *pick_next_task_ras(struct rq *rq) {
  if (unlikely(!rq->ras.ras_nr_running))
    return NULL;
  struct ras_rq *ras_rq = &rq->ras;
  struct task_struct *p;
  struct sched_ras_entity *head;

  head = list_first_entry(&rq->ras.queue, struct sched_ras_entity, run_list);
  p = container_of(head, struct task_struct, ras);
  if (!p)
    return NULL;
  p->se.exec_start = rq->clock_task;
  return p;
}
```

- requeue_task_ras: if a process has used its timeslice but not executed completely, we just move it to the tail of the running queue to ensure that it can function normally.

```
static void requeue_task_ras(struct rq *rq, struct task_struct *p, int head) {
  printk("Requeue The Task!\n");
  struct sched_ras_entity *ras_se = &p->ras;
  struct ras_rq *ras_rq = &rq->ras;
  struct list_head *queue = &rq->ras.queue;
  // move the current running task to the tail of the queue
  if (head)
    list_move(&ras_se->run_list, queue);
  else
    list_move_tail(&ras_se->run_list, queue);
}
```

- task_tick_ras: This function is the core of the scheduler. It is called periodically by the main scheduler to reduce current process's timeslice and trigger rescheduling if needed. The main task of the function is to count the left timeslice and judge whether it needs rescheduling. We alse do the computation of our timeslice according to the task's wcounts in this function.

```
static void task_tick_ras(struct rq *rq, struct task_struct *p, int queued) {
  // THE MOST IMPORTANT!!
  struct sched_ras_entity *ras_se = &p->ras;
  struct list_head *queue = &rq->ras.queue;
  update_curr_ras(rq);
  if (p->policy != SCHED_RAS)
    return;
  printk(
      "!task_tick||cpu: %d pid: %d time_slice: %d nr_running: %d wcounts: %d\n"
      ,
      cpu_of(rq), p->pid, p->ras.time_slice, rq->ras.ras_nr_running,
      p->wcounts);

  if (--p->ras.time_slice)
    return;
  // My strategy: use an inverse function to realize wcounts of [0,INF) to race
  // probability of [0,10]
  p->ras.race_prob = 11 * p->wcounts / (p->wcounts + 2000);
  // And then use a function to get timeslice of [10,100]
  p->ras.time_slice = 100 - 9 * p->ras.race_prob;
  if (ras_se->run_list.prev != ras_se->run_list.next) {
    requeue_task_ras(rq, p, 0);
    resched_task(p);
    return;
  }
}
```

Here I designed a simple method to assign some timeslices for each process. The RAS Scheduler assign more milliseconds as a time slice for tasks with lower race probabilities, which means with less page writes to the virtual memory. Because the wcounts may be very large, so I designed a map from [0,inf) to [0,10].

$$Race\ prob = \frac{11 * wcounts}{wcounts + C}.$$

Here C is a constant which depends on the scale of the data, and because the brutal division always get an integer so we use 11.

And because the race probabilities are integers. We just use a linear mapping to get the timeslice.

$$Timeslice = 100 - 9 * race\ prob$$

So in this way we assign different timeslices for different tasks according to page race. To achieve this, we only need to employ the syscall we've written in the test file.

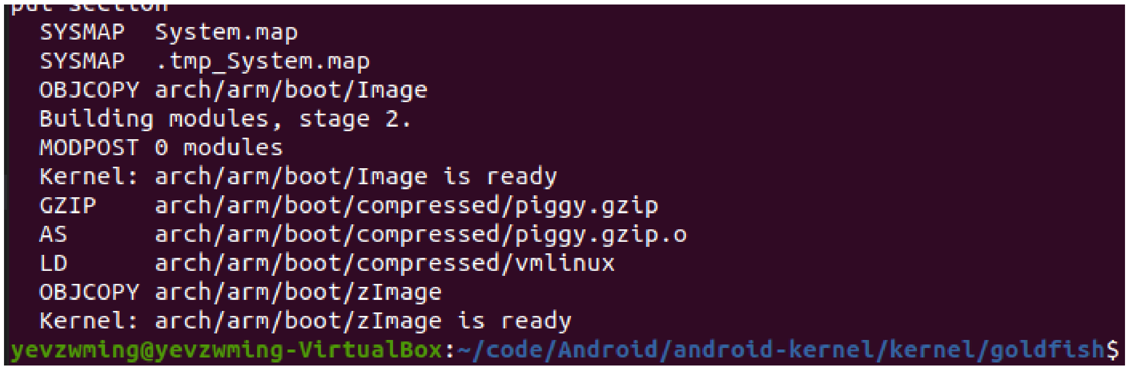
- get__rr__interval__ras: The function is to assign timeslice for each process. My understanding to this function is very simple: to return the timeslice calculated based on race probabilities.

```c
static unsigned int get__rr__interval__ras(struct rq *rq,
                                           struct task__struct *task) {
  // This function is also somewhat strange to me.
  if (!rq || !task) {
    printk("INVALID RAS!\n");
    return -1;
  }
  // return the timeslice??
  return 100 - 9 * task->ras.race__prob;
}
```

Here we've completed most of the essential functions of RAS Scheduler. Some small functions are also needed to get the pointer of some variables such as ras__rq, sched__ras__entity, task__struct and so on. These functions can be referred to in my source code.

Also we need to do some changes to the scheduler's structure to update the scheduling class in a new order: We set the .next in rt.c to be fair__sched__entity; the .next in the fair.c to be ras__sched__entity; the .next in the ras.c to be the idle__sched__entity.

So far almost all work done, with some more details considered, we open the terminal in goldfish and use the command "make -j4", and we can compile the new kernel (maybe it will take a few minutes).



```
  SYSMAP  System.map
  SYSMAP  .tmp_System.map
  OBJCOPY arch/arm/boot/Image
  Building modules, stage 2.
  MODPOST 0 modules
  Kernel: arch/arm/boot/Image is ready
  GZIP    arch/arm/boot/compressed/piggy.gzip
  AS      arch/arm/boot/compressed/piggy.gzip.o
  LD      arch/arm/boot/compressed/vmlinux
  OBJCOPY arch/arm/boot/zImage
  Kernel: arch/arm/boot/zImage is ready
yevzwming@yevzwming-VirtualBox:~/code/Android/android-kernel/kernel/goldfish$
```

Figure 1: Make the Android Kernel

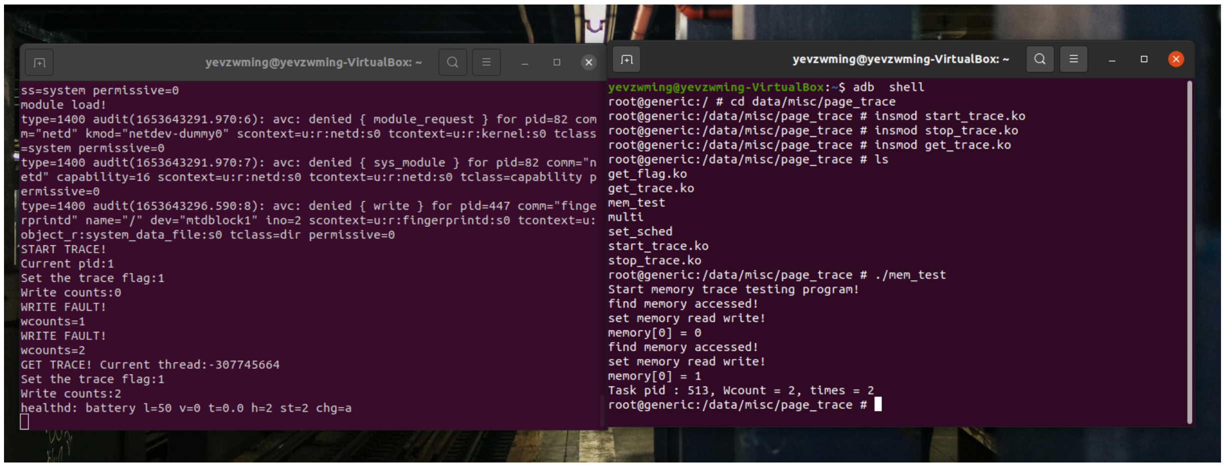
## 4 Test Results

In this section, I will list my test results for each problem in this project.

## 4.1 Page Trace Test

To test our page trace syscalls, referring to the special mechanism our TA provided, we should implement a segv_handler function to handle the page fault signal, canceling the protection for pages. And then we need to reset protection and continue our test.

And using the mem_test file provided by TA, the testing result is as figure2 depicts:



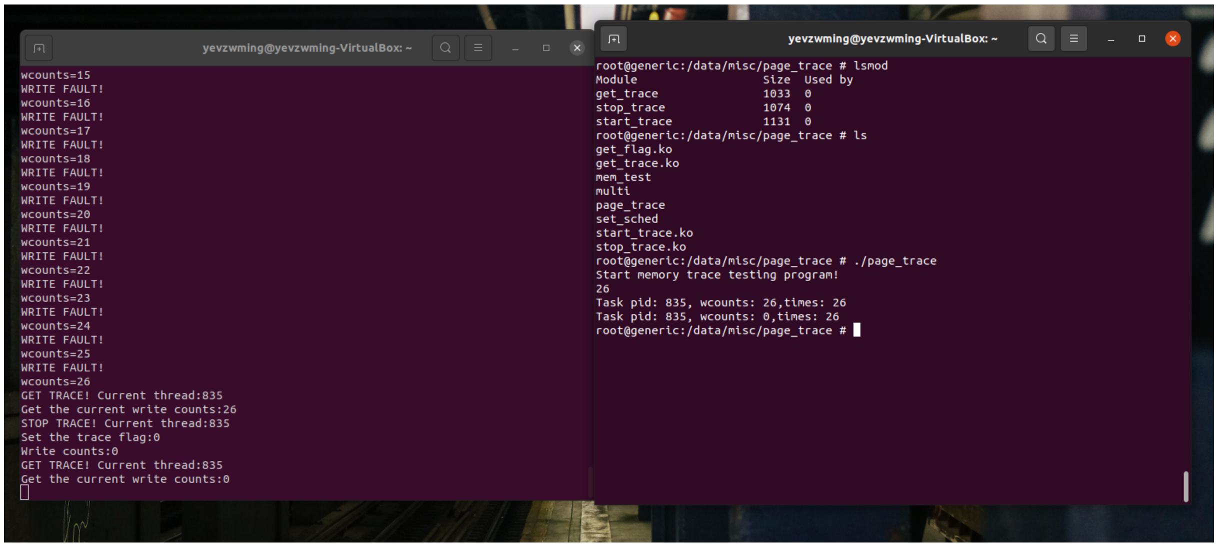
yevzwming@yevzwming-VirtualBox: ~
yevzwming@yevzwming-VirtualBox: ~

```
ss=system permissive=0
module load!
type=1400 audit(1653643291.970:6): avc: denied { module_request } for pid=82 com
m="netd" kmod="netdev-dummy0" scontext=u:r:netd:s0 tcontext=u:r:kernel:s0 tclass
=system permissive=0
type=1400 audit(1653643291.970:7): avc: denied { sys_module } for pid=82 comm="n
etd" capability=16 scontext=u:r:netd:s0 tcontext=u:r:netd:s0 tclass=capability p
ermissive=0
type=1400 audit(1653643296.590:8): avc: denied { write } for pid=447 comm="finge
rprintd" name="/" dev="mtdblock1" ino=2 scontext=u:r:fingerprintd:s0 tcontext=u:
object_r:system_data_file:s0 tclass=dir permissive=0
START TRACE!
Current pid:1
Set the trace flag:1
Write counts:0
WRITE FAULT!
wcounts=1
WRITE FAULT!
wcounts=2
GET TRACE! Current thread:-307745664
Set the trace flag:1
Write counts:2
healthd: battery l=50 v=0 t=0.0 h=2 st=2 chg=a
```

```
yevzwming@yevzwming-VirtualBox:~$ adb  shell
root@generic:/ # cd data/misc/page_trace
root@generic:/data/misc/page_trace # insmod start_trace.ko
root@generic:/data/misc/page_trace # insmod stop_trace.ko
root@generic:/data/misc/page_trace # insmod get_trace.ko
root@generic:/data/misc/page_trace # ls
get_flag.ko
get_trace.ko
mem_test
multi
set_sched
start_trace.ko
stop_trace.ko
root@generic:/data/misc/page_trace # ./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 : 513, Wcount = 2, times = 2
root@generic:/data/misc/page_trace #
```

Figure 2: Page trace test by TA's mem_test

In my own test file, I use a for loop to write to the memory for n times as n is an input integer. I also use syscall stop_trace to clear the task's wcounts. The result is as figure3 depicts:



yevzwming@yevzwming-VirtualBox: ~
yevzwming@yevzwming-VirtualBox: ~

```
wcounts=15
WRITE FAULT!
wcounts=16
WRITE FAULT!
wcounts=17
WRITE FAULT!
wcounts=18
WRITE FAULT!
wcounts=19
WRITE FAULT!
wcounts=20
WRITE FAULT!
wcounts=21
WRITE FAULT!
wcounts=22
WRITE FAULT!
wcounts=23
WRITE FAULT!
wcounts=24
WRITE FAULT!
wcounts=25
WRITE FAULT!
wcounts=26
GET TRACE! Current thread:835
Get the current write counts:26
STOP TRACE! Current thread:835
Set the trace flag:0
Write counts:0
GET TRACE! Current thread:835
Get the current write counts:0
```

```
root@generic:/data/misc/page_trace # lsmod
Module                  Size  Used by
get_trace               1033  0
stop_trace              1074  0
start_trace             1131  0
root@generic:/data/misc/page_trace # ls
get_flag.ko
get_trace.ko
mem_test
multi
page_trace
set_sched
start_trace.ko
stop_trace.ko
root@generic:/data/misc/page_trace # ./page_trace
Start memory trace testing program!
26
Task pid: 835, wcounts: 26,times: 26
Task pid: 835, wcounts: 0,times: 26
root@generic:/data/misc/page_trace #
```

Figure 3: Page trace test by myself

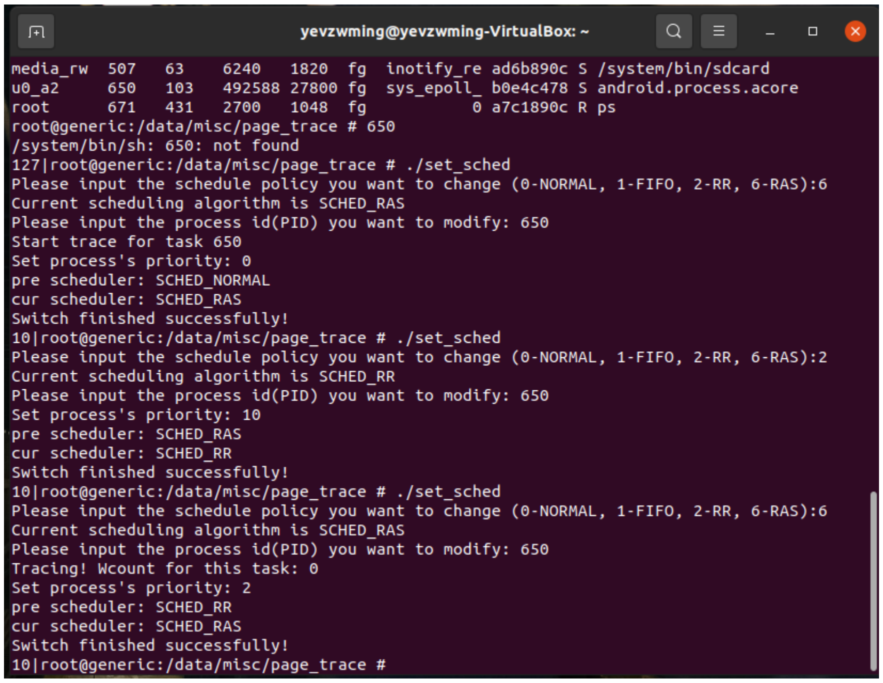
## 4.2 Change Scheduler Test

The second test is to change the scheduler in our user space. Here we use the syscall sched_-setscheduler defined in the core.c and use it to change a given pid's scheduling policy by hand.

Here because we didn't introduce the priority schemes so for real-time processes, we need to assign a priority for them; for RAS processes, we skip this stage and take it as 0.

In this section, I also add a new syscall 364 to get the status of the trace_flag of the current task_struct. If we set it as RAS as the first time we will print "Start Tracing!". Otherwise we just print the current wcounts.

After the Android device works normally, we use ps -P to choose a task to change. The result is as figure4 depicts:



```
media_rw  507   63    6240   1820  fg  inotify_re ad6b890c S /system/bin/sdcard
u0_a2     650   103   492588 27800 fg  sys_epoll_ b0e4c478 S android.process.acore
root      671   431   2700   1048  fg           0 a7c1890c R ps
root@generic:/data/misc/page_trace # 650
/system/bin/sh: 650: not found
127|root@generic:/data/misc/page_trace # ./set_sched
Please input the schedule policy you want to change (0-NORMAL, 1-FIFO, 2-RR, 6-RAS):6
Current scheduling algorithm is SCHED_RAS
Please input the process id(PID) you want to modify: 650
Start trace for task 650
Set process's priority: 0
pre scheduler: SCHED_NORMAL
cur scheduler: SCHED_RAS
Switch finished successfully!
10|root@generic:/data/misc/page_trace # ./set_sched
Please input the schedule policy you want to change (0-NORMAL, 1-FIFO, 2-RR, 6-RAS):2
Current scheduling algorithm is SCHED_RR
Please input the process id(PID) you want to modify: 650
Set process's priority: 10
pre scheduler: SCHED_RAS
cur scheduler: SCHED_RR
Switch finished successfully!
10|root@generic:/data/misc/page_trace # ./set_sched
Please input the schedule policy you want to change (0-NORMAL, 1-FIFO, 2-RR, 6-RAS):6
Current scheduling algorithm is SCHED_RAS
Please input the process id(PID) you want to modify: 650
Tracing! Wcount for this task: 0
Set process's priority: 2
pre scheduler: SCHED_RR
cur scheduler: SCHED_RAS
Switch finished successfully!
10|root@generic:/data/misc/page_trace #
```

Figure 4: Set scheduler

### 4.3 Multi-Process Racing Test

In this test part I use fork() to create n child tasks to run a large loop where each child task continues to write in a protected memory. In this case we have formed a race relationship, and we can observe how our RAS Scheduler cope with the scheduling issues in a weighted-round-robin way.

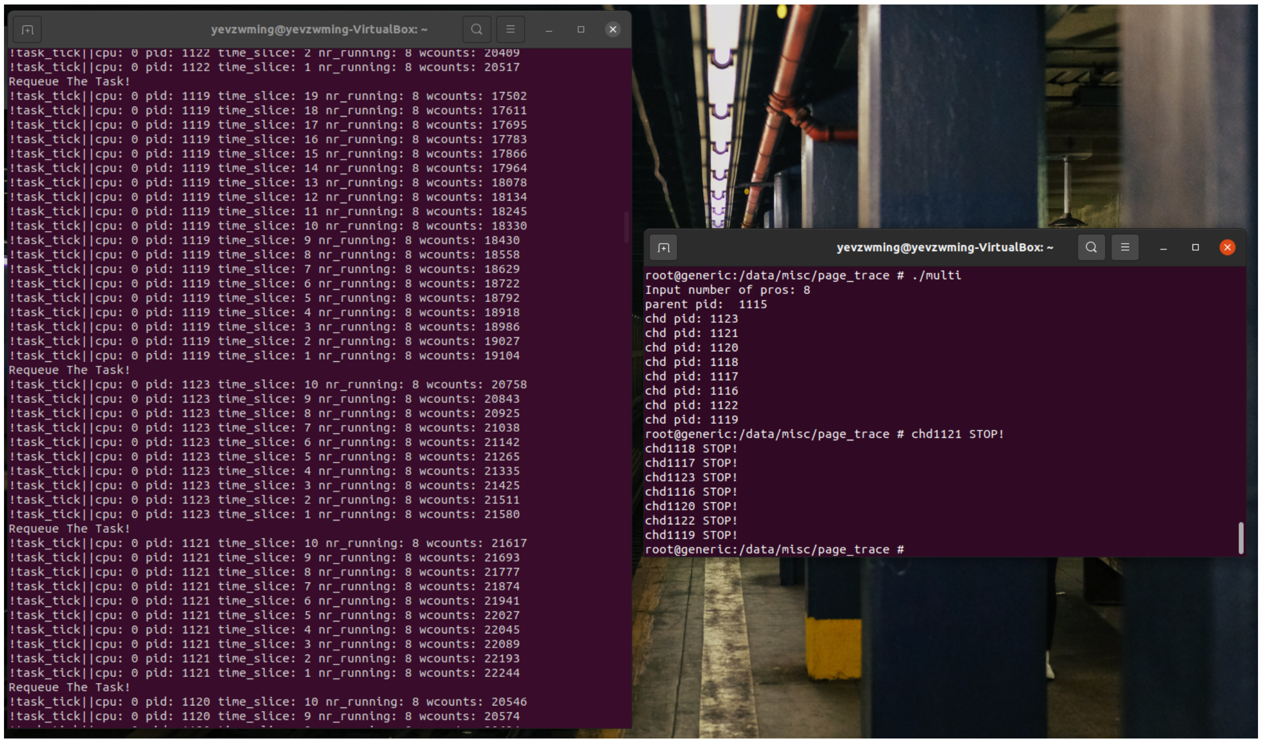
Here we define a large integer k(here we take 100000)for while looping to ensure the racing time is long enough.

Referring to the previous testing file, we allocate a size of memory for each of the child process and let them race the memory resources, counting their wcounts.

The test result is as figure5 depicts.

## 5 Obstacles and Inspirations

The core problem of this project is to implement the RAS Scheduler. To achieve this, we need to read a lot of source code in the Linux Kernel and learn about the rt scheduler to accomplish our own scheduler. It takes me almost 2 weeks to fully accomplish this project on my own. So


yevzwming@yevzwming-VirtualBox: ~
yevzwming@yevzwming-VirtualBox: ~

```
!task_tick||cpu: 0 pid: 1122 time_slice: 2 nr_running: 8 wcounts: 20409
!task_tick||cpu: 0 pid: 1122 time_slice: 1 nr_running: 8 wcounts: 20517
Requeue The Task!
!task_tick||cpu: 0 pid: 1119 time_slice: 19 nr_running: 8 wcounts: 17502
!task_tick||cpu: 0 pid: 1119 time_slice: 18 nr_running: 8 wcounts: 17611
!task_tick||cpu: 0 pid: 1119 time_slice: 17 nr_running: 8 wcounts: 17695
!task_tick||cpu: 0 pid: 1119 time_slice: 16 nr_running: 8 wcounts: 17783
!task_tick||cpu: 0 pid: 1119 time_slice: 15 nr_running: 8 wcounts: 17866
!task_tick||cpu: 0 pid: 1119 time_slice: 14 nr_running: 8 wcounts: 17964
!task_tick||cpu: 0 pid: 1119 time_slice: 13 nr_running: 8 wcounts: 18078
!task_tick||cpu: 0 pid: 1119 time_slice: 12 nr_running: 8 wcounts: 18134
!task_tick||cpu: 0 pid: 1119 time_slice: 11 nr_running: 8 wcounts: 18245
!task_tick||cpu: 0 pid: 1119 time_slice: 10 nr_running: 8 wcounts: 18330
!task_tick||cpu: 0 pid: 1119 time_slice: 9 nr_running: 8 wcounts: 18430
!task_tick||cpu: 0 pid: 1119 time_slice: 8 nr_running: 8 wcounts: 18558
!task_tick||cpu: 0 pid: 1119 time_slice: 7 nr_running: 8 wcounts: 18629
!task_tick||cpu: 0 pid: 1119 time_slice: 6 nr_running: 8 wcounts: 18722
!task_tick||cpu: 0 pid: 1119 time_slice: 5 nr_running: 8 wcounts: 18792
!task_tick||cpu: 0 pid: 1119 time_slice: 4 nr_running: 8 wcounts: 18918
!task_tick||cpu: 0 pid: 1119 time_slice: 3 nr_running: 8 wcounts: 18986
!task_tick||cpu: 0 pid: 1119 time_slice: 2 nr_running: 8 wcounts: 19027
!task_tick||cpu: 0 pid: 1119 time_slice: 1 nr_running: 8 wcounts: 19104
Requeue The Task!
!task_tick||cpu: 0 pid: 1123 time_slice: 10 nr_running: 8 wcounts: 20758
!task_tick||cpu: 0 pid: 1123 time_slice: 9 nr_running: 8 wcounts: 20843
!task_tick||cpu: 0 pid: 1123 time_slice: 8 nr_running: 8 wcounts: 20925
!task_tick||cpu: 0 pid: 1123 time_slice: 7 nr_running: 8 wcounts: 21038
!task_tick||cpu: 0 pid: 1123 time_slice: 6 nr_running: 8 wcounts: 21142
!task_tick||cpu: 0 pid: 1123 time_slice: 5 nr_running: 8 wcounts: 21265
!task_tick||cpu: 0 pid: 1123 time_slice: 4 nr_running: 8 wcounts: 21335
!task_tick||cpu: 0 pid: 1123 time_slice: 3 nr_running: 8 wcounts: 21425
!task_tick||cpu: 0 pid: 1123 time_slice: 2 nr_running: 8 wcounts: 21511
!task_tick||cpu: 0 pid: 1123 time_slice: 1 nr_running: 8 wcounts: 21580
Requeue The Task!
!task_tick||cpu: 0 pid: 1121 time_slice: 10 nr_running: 8 wcounts: 21617
!task_tick||cpu: 0 pid: 1121 time_slice: 9 nr_running: 8 wcounts: 21693
!task_tick||cpu: 0 pid: 1121 time_slice: 8 nr_running: 8 wcounts: 21777
!task_tick||cpu: 0 pid: 1121 time_slice: 7 nr_running: 8 wcounts: 21874
!task_tick||cpu: 0 pid: 1121 time_slice: 6 nr_running: 8 wcounts: 21941
!task_tick||cpu: 0 pid: 1121 time_slice: 5 nr_running: 8 wcounts: 22027
!task_tick||cpu: 0 pid: 1121 time_slice: 4 nr_running: 8 wcounts: 22045
!task_tick||cpu: 0 pid: 1121 time_slice: 3 nr_running: 8 wcounts: 22089
!task_tick||cpu: 0 pid: 1121 time_slice: 2 nr_running: 8 wcounts: 22193
!task_tick||cpu: 0 pid: 1121 time_slice: 1 nr_running: 8 wcounts: 22244
Requeue The Task!
!task_tick||cpu: 0 pid: 1120 time_slice: 10 nr_running: 8 wcounts: 20546
!task_tick||cpu: 0 pid: 1120 time_slice: 9 nr_running: 8 wcounts: 20574
```

```
root@generic:/data/misc/page_trace # ./multi
Input number of pros: 8
parent pid:  1115
chd pid: 1123
chd pid: 1121
chd pid: 1120
chd pid: 1118
chd pid: 1117
chd pid: 1116
chd pid: 1122
chd pid: 1119
root@generic:/data/misc/page_trace # chd1121 STOP!
chd1118 STOP!
chd1117 STOP!
chd1123 STOP!
chd1116 STOP!
chd1120 STOP!
chd1122 STOP!
chd1119 STOP!
root@generic:/data/misc/page_trace #
```

Figure 5: Multi-Process Racing

meanwhile I met a lot of obstacles and dedicate a lot to find inspirations to solve them. Here are some representative ones:

When I implement the page trace mechanism, at first I revised the do_page_fault function in kernel/sched/fault.c to match the condition of our expected write fault. At first I added the judging condition: if(tsk->tracing&&(fsr& FSR_WRITE)). That's not correct and my wcounts are larger than normal. So I attempted to revise the access_error function: judge whether it is a write operation and continue analyzing virtual memory access, according to the vm_flags. In this way I succeeded getting the right outcome.

And the compilation of the kernel after designing RAS Scheduler is also a tough task. At first I implement most of the necessary functions according to those in rt.c. But rt.c is too complicated for our simple scheduler. Analyzing the error information, I found that most of the errors were associated with the priority. But I read the instructions on the ppt again and found that it could be all right even if we do not use priorities. So I simplified the model, removing the prio_array, bitmap structures, and so on.

Another obstacle is when I implement the set_sched test file. And this is also related with the prio problem, because the the rt, fair and our ras tasks have different schemes for prios. At first I didn't add any operations concerning the prio. But the kernel always returned EINVAL. By debugging the error information I located the problem in the priority judgement in the ___sched_-setscheduler function in core.c.

```
if (param->sched_priority < 0 ||(p->mm && param->sched_priority >
    MAX_USER_RT_PRIO-1) || (!p->mm && param->sched_priority > MAX_RT_PRIO-1))
    return -EINVAL;
if (rt_policy(policy) != (param->sched_priority != 0))
    return -EINVAL;
```

And I add something regulating the priorities in the test file:

```
if(policy!=6)
    param.sched_priority = tmp;
else
    param.sched_priority = 0;
```

And finally I succeeded switching the schedulers for tasks.

The last torturing problem is the multi-task fork. At first I misunderstood the fork() function, and let the child processes continue to fork child processes, spelling chaos. And after searching some useful information of the use of fork, I learnt how to create exactly n child processes. But I still failed to get the tasks running normally as the nr_running always stayed 1. It seemed as if one process monopolized the scheduler and other child processes could not get enqueued. After discussing with my TA Hang Zeng, I realized that the parent task kept being preempted and could not fork other child processes. That's the key! And I set my list of scheduling classes in a correct order, and my RAS Scheduler worked in a normal state finally.

## 6 Acknowledge

I would like to express my sincere gratitude to everyone who has helped me during my development.

Thanks to Professor Fan Wu for his well-prepared teaching and tender guidance during the whole semester.

Thanks to TA Hang Zeng for his patient help to my work.

Thanks to my friends who discuss with me and answer my questions for their patience and kindness.

## 7 Reference

- https://blog.csdn.net/weixin_42462202/article/details/102887008
- https://helix979.github.io/jkoo/post/os-scheduler/
- https://blog.csdn.net/SunnyBeiKe/article/details/6950191