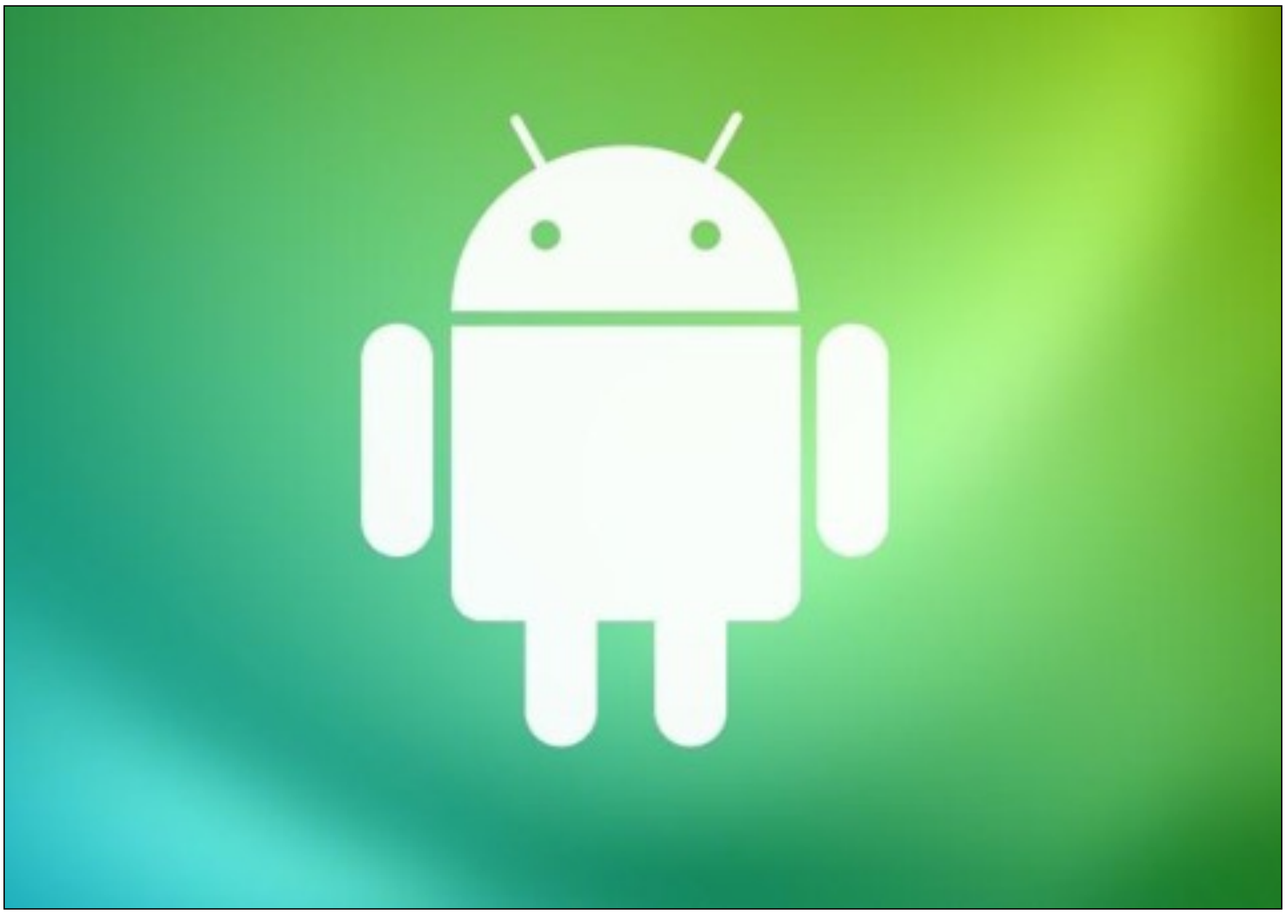
Report for OsPrj2

Android Scheduler

Lingkun Kong, 5140219016 - June 23, 2016



Introduction

1、The requirements of the OS Project 2

In general, this Project asks us to complete two kind of assignments.

One of the assignments for us is to change the scheduler of process in user's layer, in other word, it require us to write code in C, and then use ndk-build to generate the executable file in the android equipment in order to change the scheduler of process, such as test-apk given by T.A. and the descendants of the zygote process.

Another assignment ask us to modify the kernel in operating system's layer. In this case, we can initial the OS's original scheduling method. For instance, we can initial specific process's scheduler, such as setting the zygote to FIFO. And if we do this, there is no need for us to change the process's scheduler in a 'clumsy' way.

2、The structure of my report

My project report is divided into three parts. First part of my report, as you can see, is the introduction of the project and my first primary impression of the project.

And in second part, you will see my solution to the project. It will include methods I using to meet the project's requirements, and how I find these methods. Besides, I will also present some of my code in this part and show why these codes are useful.

In third part, I will show you my project results and my analysis of the results.

Methods & Codes

1、Change the scheduler in user's layer

The tutorial PDF tells us that we need to do following things: Changing single specific process's (test application) scheduler to FIFO or RR, changing several processes' (test applications) scheduler to FIFO or RR, change a bunch of processes' (all descendants of zygote) scheduler to RR, then observing the test application's running time, which is offered by T.A., as the final result.

For these requirements, I write two program in C, attached with jni file, and then build in Terminal to generate executive files which help me to change the scheduler in ADB machine.

Following is my first program I use to change single process's scheduler:

```
#include …
int main(int argc, char const *argv[])
{
  struct sched_param param;
  int myScheduler;
  printf("Please input the Scheduling method (0-normal,1-FIFO,2-RR): “);
// input the Scheduling method you want to use
  scanf("%d",&myScheduler);
  switch(myScheduler)
  {
            case 0 : printf("Current scheduling method is SCHED_Normal\n");break;
            case 1 : printf("Current scheduling method is SCHED_FIFO\n");break;
            case 2 : printf("Current scheduling method is SCHED_RR\n");break;
            default: printf("Invalid input.\n"); return -1;break;
  }

  int myPri;
  int processpid;
  printf("Please input the id of the testprocess : “);
// input the id of the Process you want change its scheduler
  scanf("%d",&processpid);
  if (myScheduler == 0) //  sched_get_priority_max(0) only can be zero
  {
            myPri = 0;
  }
  else
  {
          printf("Set Process's priority (1-99): “);
// input the priority you want your new scheduler has
          scanf("%d", &myPri);
          if (myPri < 1 || myPri > 99)
          {
                  perror("Priority level doesn't exist.");
                  return -1;
          }
  }
// tell user the result of his operation
  printf("current scheduler's priority is : ");
  printf("%d\n",myPri);
  printf("pre scheduler : ");
  printf("%d\n", sched_getscheduler(processpid));
  param.sched_priority = myPri;
// shced_setscheduler changes the scheduler of Process
  if (sched_setscheduler(processpid, myScheduler, &param) == -1)
  {
       perror("sched_setscheduler() failed");
       return -1;
  }
  printf("cur scheduler : ");
  printf("%d\n", sched_getscheduler(processpid));
  return 0;
}
```

The main goal of the code above is to let user can tell computer which process he want to change, the parameter of the new scheduler and can see what happens to the process user picks.

In fact, the core of this program is the usage of ***sched_setscheduler(processpid, myScheduler, &param)*** function. The definition of these function is in <sched.h>:

```
int sched_setscheduler(pid_t pid, int policy, const struct sched_param
*param);
int sched_getscheduler(pid_t pid);
struct sched_param { …
          int sched_priority;
          … };
```

By the definition of this function, we can easily set the scheduler of a process if we know the process's id and set the priority in 'struct sched_param'.

But how can we know the pid of the specific process? According to the PDF's recommendation, I use ps -p command in shell to see the process's id and its scheduling info. And we can also check whether the new scheduler's setting is all right.

So far I have succeeded in set single process's scheduler. But how to set multiple processes' scheduler. In fact, the code is almost as same as the code above. What I need to do is to let user input multiple processes' id.

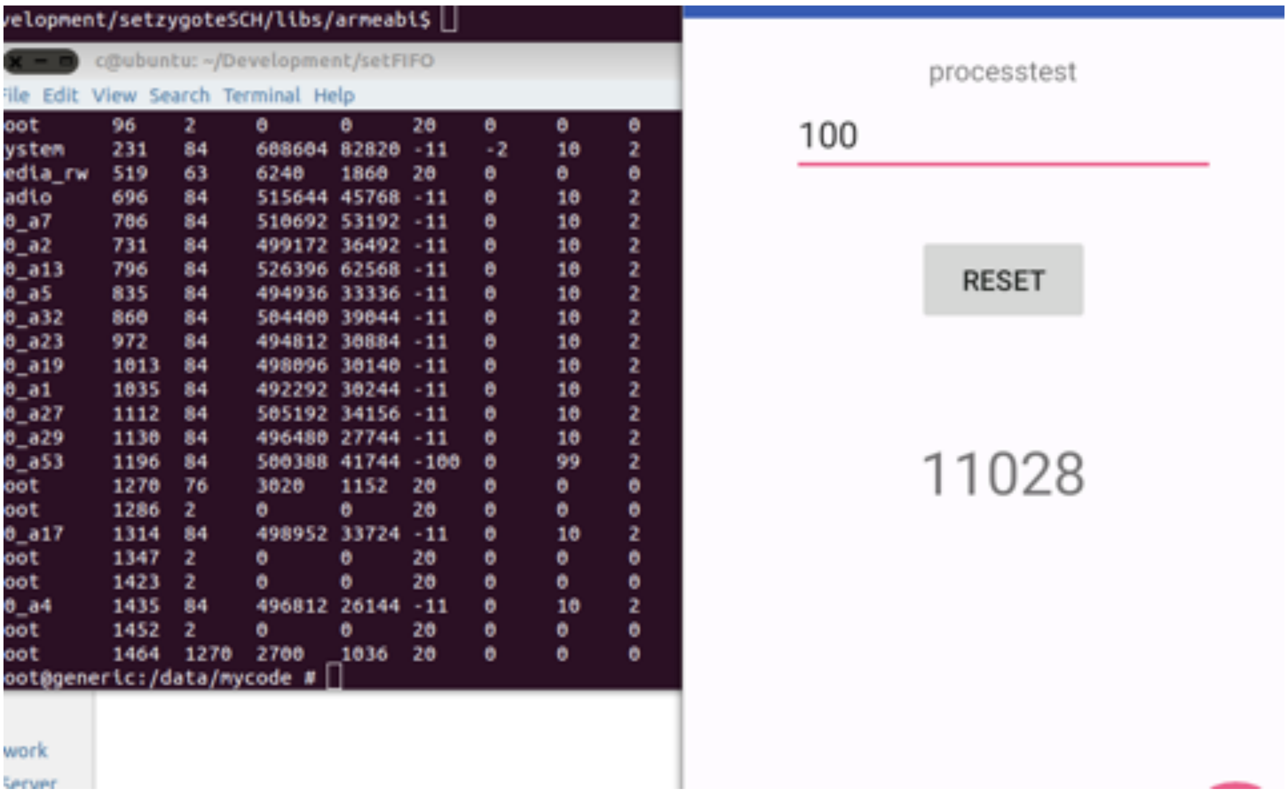
The code is like this:

```
int main(int argc, char const *argv[])
{
  struct sched_param param;
  int myScheduler;
  printf("Please input the Scheduling method (0-normal,1-FIFO,2-RR): ");
  scanf("%d",&myScheduler);
  switch(myScheduler)
  {
            case 0 : printf("Current scheduling method is SCHED_Normal\n");break;
            case 1 : printf("Current scheduling method is SCHED_FIFO\n");break;
            case 2 : printf("Current scheduling method is SCHED_RR\n");break;
            default: printf("Invalid input.\n"); return -1;break;
  }

  int myPri;
  // input the id of processes which we want to operate
  int i = 0;
  int processpid[100];
  while(1)
  {
            printf("Please input the id of the testprocess : (quit by input -1)");
            scanf("%d",&processpid[i]);
            // when input -1, program begin to set scheduler and exit reading
            if (processpid[i]==-1)
            {
                      break;
            }
            i ++ ;
  }
  // set Priority
  if (myScheduler == 0) //  sched_get_priority_max(0) only can be zero
  {
            myPri = 0;
  }
  else
  {
            printf("Set Process's priority (1-99): ");
            scanf("%d", &myPri);
            if (myPri < 1 || myPri > 99)
            {
                      perror("Priority level doesn't exist.");
                      return -1;
            }
  }
```

```
    printf("current scheduler's priority is : ");
    printf("%d\n",myPri);
    printf("pre scheduler : ");
    printf("%d\n", sched_getscheduler(processpid[0]));
    param.sched_priority = myPri;
  // using for loop to set scheduler
    for (i = i -1 ; i >= 0; --i) // i = i -1 exclude -1
    {
            if (sched_setscheduler(processpid[i], myScheduler, &param) == -1)
            {
            perror("sched_setscheduler() failed");
            return -1;
            }
    }
    printf("cur scheduler : ");
    printf("%d\n", sched_getscheduler(processpid[0]));
    return 0;
  }
```

I just use a for loop to set the process's scheduler. And by these simple and intuitive method, we can easily set a bunch of processes' schedulers.

And when I use ps -p command to see whether the code is correct, I get this picture showing below , which is about setting all of zygote's (pid : 84) descendants' scheduler to RR, and set the test app's (pid:1196) priority to 99 while others to 10.



```
elopment/setzygoteSCH/libs/armeabi$
```

```
oot       96    2     0       0      20    0    0    0
ystem     231   84    608604  82820  -11   -2   10   2
edia_rw   519   63    6240    1860   20    0    0    0
adio      696   84    515644  45768  -11   0    10   2
0_a7      786   84    510692  53192  -11   0    10   2
0_a2      731   84    499172  36492  -11   0    10   2
0_a13     796   84    526396  62568  -11   0    10   2
0_a5      835   84    494936  33336  -11   0    10   2
0_a32     860   84    504400  39044  -11   0    10   2
0_a23     972   84    494812  30884  -11   0    10   2
0_a19     1013  84    498096  30140  -11   0    10   2
0_a1      1035  84    492292  30244  -11   0    10   2
0_a27     1112  84    505192  34156  -11   0    10   2
0_a29     1130  84    496480  27744  -11   0    10   2
0_a53     1196  84    500388  41744  -100  0    99   2
oot       1270  76    3020    1152   20    0    0    0
oot       1286  2     0       0      20    0    0    0
0_a17     1314  84    498952  33724  -11   0    10   2
oot       1347  2     0       0      20    0    0    0
oot       1423  2     0       0      20    0    0    0
0_a4      1435  84    496812  26144  -11   0    10   2
oot       1452  2     0       0      20    0    0    0
oot       1464  1270  2700    1036   20    0    0    0
oot@generic:/data/mycode #
```

c@ubuntu: ~/Development/setFIFO
File Edit View Search Terminal Help
work
Server
processtest
100
RESET
11028

2、Modify the kernel

The problem3 of the tutorial PDF needs us to set the Default scheduler of all descendants of process zygote to RR, and priority to a number decided by the id of the process. Besides, the problem3 also wants us to change the policy of the Sched_RR. And specifically , it needs RR to pick the next process randomly.

But how can we modify the kernel? And where should we add or detract our own code?

By the prompt of the PDF, I find that I should first well understand the rt.c and the core.c file in kernel file, and only when I figure out the proceeding of the Android tasks setting schedulers, I can write the correct code for the problem3's requirement.

Now, I want to show you how I solve the problem3.

As a matter of fact, the problem3, as the tutorial PDF hints, needs me to change the "core.c" and "rt.c" file. And in the meanwhile, I read the "shed.h" as reference and find functions' definition by Internet.And I find that the function "fork.c" should also be modified. After I do all of these, the problem3 seems to be kind of easy.

a、Solve the problem 3-1, set default scheduler

First of all, let me show you how to set default scheduler for all descendants of process zygote. In this part of problems, I modify the "core.c" file after code as "sched_reset_on_fork", line 1773. Besides, I modify the "fork.c" file after in function "copy_process()", line 1471.

At the very beginning, I in fact want to change the code right in the core.c's "sched_fork()" function, such as every time when I fork a process, I check whether it is the descendant of zygote and then set the scheduler's property. However, I soon realised that after the sched_fork() function, the process in fact doesn't have its on pid, and the its pid is inherit from their parents. So when we use 'ps -p' command, what we see is that the new process's priority is as same as the zygote's.

In this situation, I change the default scheduler mode (RR) in core.c because it is no need to use pid info, and set the scheduler's priority in fork.c when the child process get its name.

Below is the modified code in core.c, and the red part is the added code in core.c :

```
……
if (unlikely(p->sched_reset_on_fork)) {
        if (task_has_rt_policy(p)) {
                p->policy = SCHED_NORMAL;
                p->static_prio = NICE_TO_PRIO(0);
                p->rt_priority = 0;
        } else if (PRIO_TO_NICE(p->static_prio) < 0)
                p->static_prio = NICE_TO_PRIO(0);

        p->prio = p->normal_prio = __normal_prio(p);
        set_load_weight(p);

        /*
         * We don't need the reset flag anymore after the fork. It has
         * fulfilled its duty:
         */
    // This if statement is used to reset the scheduler of the process.
    // So that is the reason why I add my code after the resetting
        p->sched_reset_on_fork = 0;
    }
   // My code
    if( strcmp(p->comm,"main") == 0)
        p ->policy = SCHED_RR;
```

Below is the modified code in fork.c, and the red part is the added code in fork.c :

```
……
// this part means the process get its unique id but we cannot add code right after process get its pid because now the process still hasn't built its parent structure
p->pid = pid_nr(pid);
p->tgid = p->pid;
……
// this part means the process builds its structure in linux, so only after that, we can use pointer to point the process's parent or real_parent
if (clone_flags & (CLONE_PARENT|CLONE_THREAD)) {
        p->real_parent = current->real_parent;
        p->parent_exec_id = current->parent_exec_id;
 } else {
        p->real_parent = current;
        p->parent_exec_id = current->self_exec_id;
 }
 //add code here, 100 is the max priority by system's definition
 if( strcmp(p->real_parent->comm,"main") == 0)
        p ->rt_priority = 100 / 5 * (p->pid % 5) + 1;
```

One thing that is a kind of tricky is that when the “zygote” calls for fork command, its comm is something named like “app_process”. And only when zygote's descendants call for fork, its original comm by defaulting is “main”.

So there is no need to worry the priority of zygote is also changed.

b、Solve the problem 3-2, change the policy of RR

The problem 3-2 ask me to modify the kernel to let the RR pick the next process randomly.

First of all, there exist a class for scheduling:

```
const struct sched_class rt_sched_class = {
……
.pick_next_task  =  pick_next_task_rt,
……};
```

Pick_next_task_rt is an important parameter for my project in *rt_sched_class*, decides which process to enter the queue to get the resource of CPU.And the function *pick_next_task_rt()* invokes function *pick_next_rt_entity()* to choose the next schedule entity.

So in this part of problems, I modify the "rt.c" file in function "pick_next_rt_entity", line 1328.

Let's see the original code first:

```
static struct sched_rt_entity *pick_next_rt_entity(struct rq *rq,
                                                   struct rt_rq *rt_rq)
{
        struct rt_prio_array *array = &rt_rq->active;
        struct sched_rt_entity *next = NULL;
        struct list_head *queue;
        int idx;
// the idx is in fact is to find the priority of the of the task in queue
        idx = sched_find_first_bit(array->bitmap);
        BUG_ON(idx >= MAX_RT_PRIO);
/* uses priority as offset and add it to the head of the first ready queue.
   After that, queue has pointed to the non-null ready queue with the highest
priority. */
        queue = array->queue + idx;
/* we need to change code after the list_entry() so we change the original
one-by-one module to random module*/
        next = list_entry(queue->next, struct sched_rt_entity, run_list);
        return next;

 }
```

As the comment above, we can change the policy of the sched_RR to pick the next process randomly by changing the "next" pointer.

And I use the *get_random_bytes();* function to generate the real random number to set the "next" in random position of the queue.

P.S. this function is in <linux/random.h>

The modified code is presented as below:

```
……
next = list_entry(queue->next, struct sched_rt_entity, run_list);
 // code for 3-3 is added as below
 // my_ptr is the *task_struct (Type cast)
 struct list_head *Myhead;
 struct task_struct* my_ptr;
// container_of() makes the my_ptr become 'next"s task_struct struct
 my_ptr = container_of(next, struct task_struct, rt);

 int Num_of_Task;
 int i;
 int random_num;
 // Because of the instruct of the 3-2, we only need to concern about the SCHED_RR's
situation
 if(my_ptr->policy != SCHED_RR)
         return next;
 else
 {
         // queque is the round-queque
         // I want to use Num_of_Task to present how many tasks in queque
         Myhead = queue;
         // This do-while loop make sure that next we get should be RR
         do
         {
                 queue = queue->next;
                 Num_of_Task = 0;
                 while(queue != Myhead)
                 {
                         queue = queue -> next;
                         Num_of_Task++;
                 }
                 // generate a random number in kernel
                 get_random_bytes(&random_num, sizeof(random_num));
                 random_num = random_num % Num_of_Task;

                 queue = Myhead -> next;
                 for (i = 0; i < random_num; ++i)
                 {
                         queue = queue -> next;
                 }
                 next = list_entry(queue,struct sched_rt_entity, run_list);
                 my_ptr = container_of(next, struct task_struct, rt);

                 queue = Myhead;
         // give another chance when next is not RR
         }while(my_ptr->policy != SCHED_RR);
 }
 return next;}
```

There is also one thing we should notice. That is the do-while loop.

In fact the queue in which the processes' priorities are equal, can not make sure its all element(process) own same kind of scheduler, such as some of them are FIFO, and some of them are RR while all of their priorities are 99.

So I use the do-while loop to offer chances for computer to select "next" again when "next"'s policy is FIFO or something else.

But it also generates a problem that if the in the same priority queue only few RR task exists. The for-while loop may take a lot of time to find the RR process.

But considering the problem3-1's default setting, that almost every not RR process is set to be 'normal' scheduler and 'priory 0',in other word there is no so many "other" processes in a same queue. So this worry can be ignored.

c、Contenders generated in Android Studio

The problem3 also needs me to write my own android application as contenders to compete with the test application(by T.A.). So I use android studio to write a simple android application as my contenders.

And here is my android part code:

i) main layout's xml code:

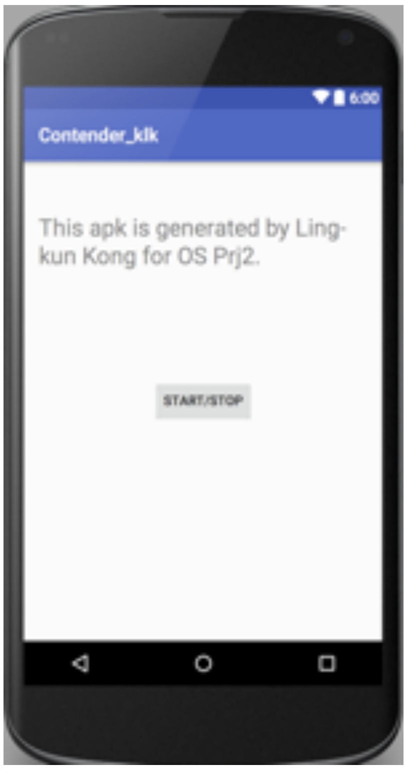
```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:paddingBottom="@dimen/activity_vertical_margin"
   android:paddingLeft="@dimen/activity_horizontal_margin"
   android:paddingRight="@dimen/activity_horizontal_margin"
   android:paddingTop="@dimen/activity_vertical_margin"
   tools:context="cn.edu.sjtu.contender_klk.ContenderTest">

   <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="This apk is generated by Ling-kun Kong for OS Prj2."
      android:visibility="visible"
      android:textSize="25dp"
      android:layout_marginTop="40dp"
      android:id="@+id/output" />

   <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
```

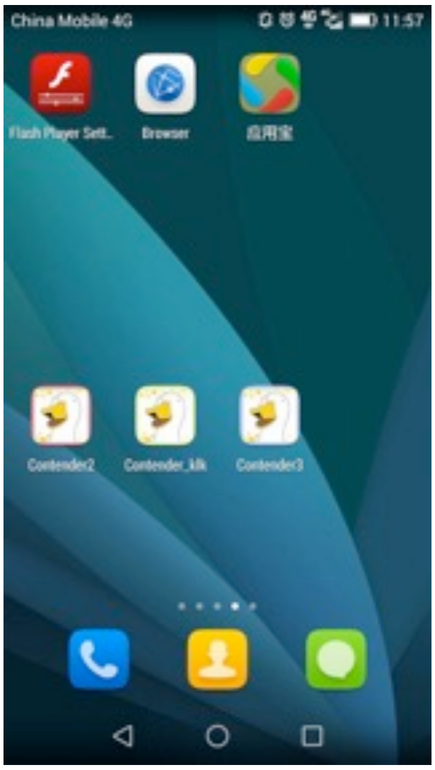
```
        android:text="Start/Stop"
        android:id="@+id/Start"
        android:layout_centerVertical="true"
        android:layout_centerHorizontal="true"
        android:onClick="Btn_Listener" />
</RelativeLayout>
```

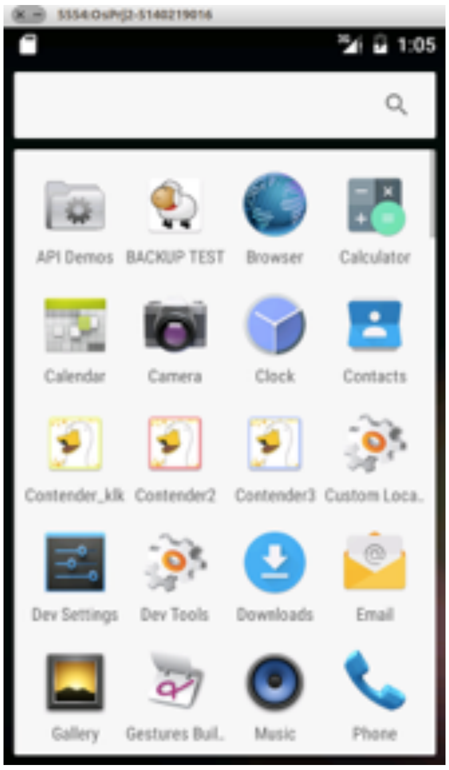
ii) main java:

```
public class ContenderTest extends AppCompatActivity {

    Button mButton;
    TextView mTextView;
    int Num = 0;
    Boolean flag = false;
    Thread newThread;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_contender_test);
        mButton = (Button) findViewById(R.id.Start);
        mTextView = (TextView) findViewById(R.id.output);
    }
    public void Btn_Listener(View view)
    {
        flag = !flag;
        String mString;
        if (flag)
        {
            mString = "Calculating ... ";
        }
        else
        {
            mString = "Run Over";
        }
        mTextView.setText(mString);
        if(flag)
        {
            newThread = new Thread(new Runnable() {
                @Override
                public void run() {
                    while (flag)
                    {
                        Num ++;
                        Num --;
                    }
                }
            });
            newThread.start();
        }
    }
}
```

And here is my apps picture:







d、Contenders generated in C

According to the requirement by T.A. , I also write my own code in C and run it as contender and test application to see the different between the real apk and arm file.

The code is offered below:

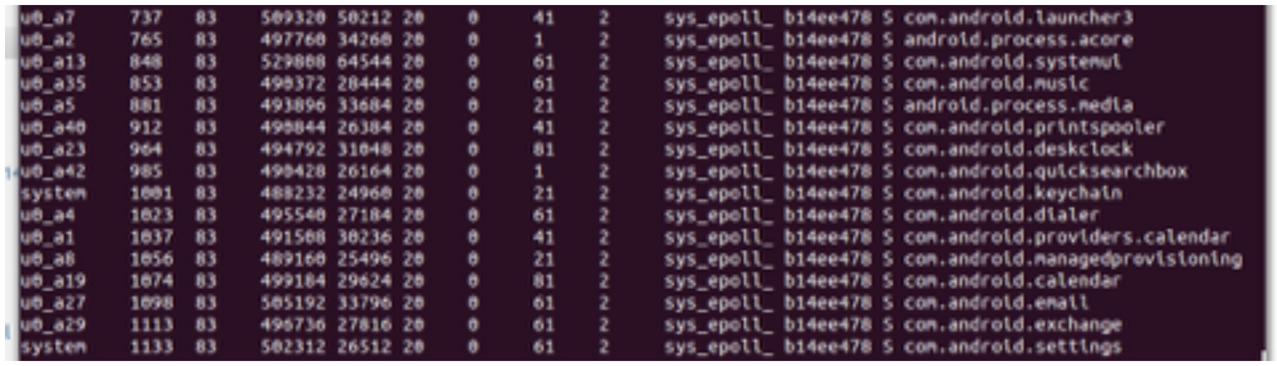
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

//calculate n!
void calc_permulate(int n)
{
  int i = 0;
  volatile   long long  res  = 1;
  for(i = n; i > 0; i--)
  {
          res  = res * n;
  }
}
```

```
int main()
{
  struct timeval time_start,time_pend;
  float finalusage;
  int n = 0;
  int  getinput = 0;

  while(1){

    printf("Please input one large number (-1 to quit) :\n");
    if(getinput == 0)  scanf("%d", &n);
    if(n == -1) break;
    if(n >= 1)
    {
          getinput = 1;
                          gettimeofday(&time_start,0);
                          calc_permulate(n);
                          gettimeofday(&time_pend,0);
// calculate the time usage
                          finalusage = 1000000 * (time_pend.tv_sec-time_start.tv_sec) +
time_pend.tv_usec - time_start.tv_usec ;
                          finalusage /= 1000;
                          printf("used time: %f ms\n",finalusage);
                          //exit(0);
      }

      n = 0;
      getinput = 0;
  }

  return 0;
}
```

In fact this program is very simple and easy to understand.

The main idea is to take advantage of the n! calculating, recording the time usage and then presenting to user.

Here is the scene after I use ps -p command when everything is settled down:



```
u0_a7     737   83    509320 50212 20    0     41    2     sys_epoll_ b14ee478 S com.android.launcher3
u0_a2     765   83    497760 34260 20    0     1     2     sys_epoll_ b14ee478 S android.process.acore
u0_a13    848   83    529888 64544 20    0     61    2     sys_epoll_ b14ee478 S com.android.systemui
u0_a35    853   83    498372 28444 20    0     61    2     sys_epoll_ b14ee478 S com.android.music
u0_a5     881   83    493896 33684 20    0     21    2     sys_epoll_ b14ee478 S android.process.media
u0_a40    912   83    498844 26384 20    0     41    2     sys_epoll_ b14ee478 S com.android.printspooler
u0_a23    964   83    494792 31048 20    0     81    2     sys_epoll_ b14ee478 S com.android.deskclock
u0_a42    985   83    498428 26164 20    0     1     2     sys_epoll_ b14ee478 S com.android.quicksearchbox
system    1001  83    488232 24960 20    0     21    2     sys_epoll_ b14ee478 S com.android.keychain
u0_a4     1023  83    495540 27184 20    0     61    2     sys_epoll_ b14ee478 S com.android.dialer
u0_a1     1037  83    491508 30236 20    0     41    2     sys_epoll_ b14ee478 S com.android.providers.calendar
u0_a8     1056  83    489160 25496 20    0     21    2     sys_epoll_ b14ee478 S com.android.managedprovisioning
u0_a19    1074  83    499184 29624 20    0     81    2     sys_epoll_ b14ee478 S com.android.calendar
u0_a27    1098  83    505192 33796 20    0     61    2     sys_epoll_ b14ee478 S com.android.email
u0_a29    1113  83    496736 27816 20    0     61    2     sys_epoll_ b14ee478 S com.android.exchange
system    1133  83    502312 26512 20    0     61    2     sys_epoll_ b14ee478 S com.android.settings
```

Result & Analysis

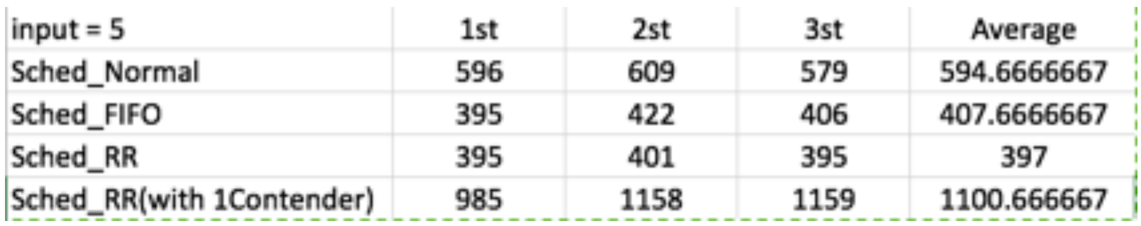
1、The result of the OS Project 2

In this project, in order to figure out the feature of different scheduler in linux, I have run plenty of experiments on many aspects.

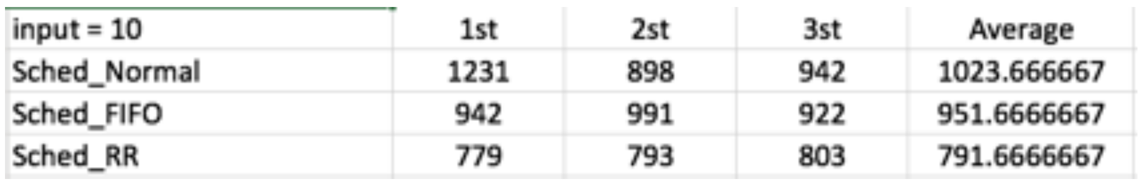
Now, let me show you the result of my experiments in the following tables.

a) **Only change the scheduler of test application offered by T.A. or with the contender** (the contender's scheduler is as same as the test application)

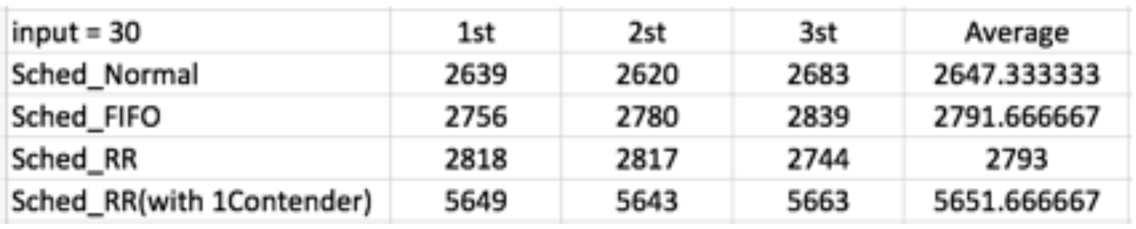
P.S. the input means the number I input into the T.A.'s application.



input = 5	1st	2st	3st	Average
Sched_Normal	596	609	579	594.6666667
Sched_FIFO	395	422	406	407.6666667
Sched_RR	395	401	395	397
Sched_RR(with 1Contender)	985	1158	1159	1100.666667



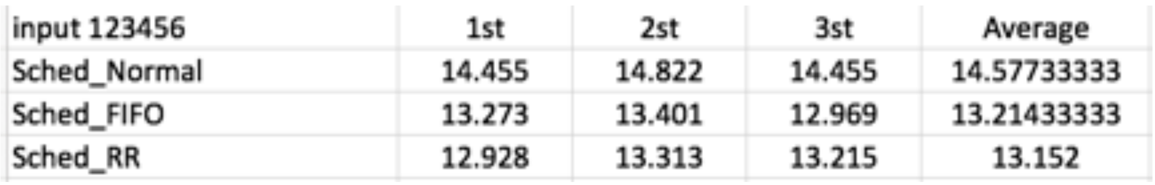
input = 10	1st	2st	3st	Average
Sched_Normal	1231	898	942	1023.666667
Sched_FIFO	942	991	922	951.6666667
Sched_RR	779	793	803	791.6666667



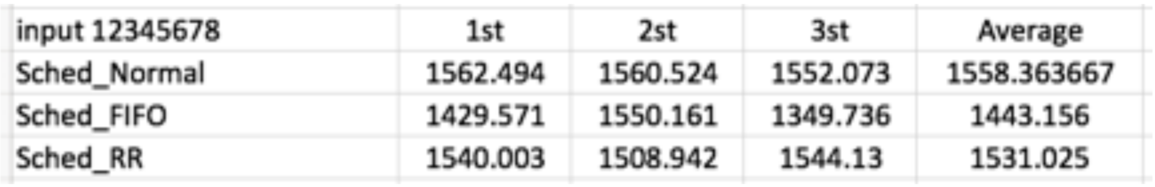
input = 30	1st	2st	3st	Average
Sched_Normal	2639	2620	2683	2647.333333
Sched_FIFO	2756	2780	2839	2791.666667
Sched_RR	2818	2817	2744	2793
Sched_RR(with 1Contender)	5649	5643	5663	5651.666667

b) Only change the scheduler of my own test application — calculating the input number's permutation. (num !)

P.S. measured in ms (millisecond)

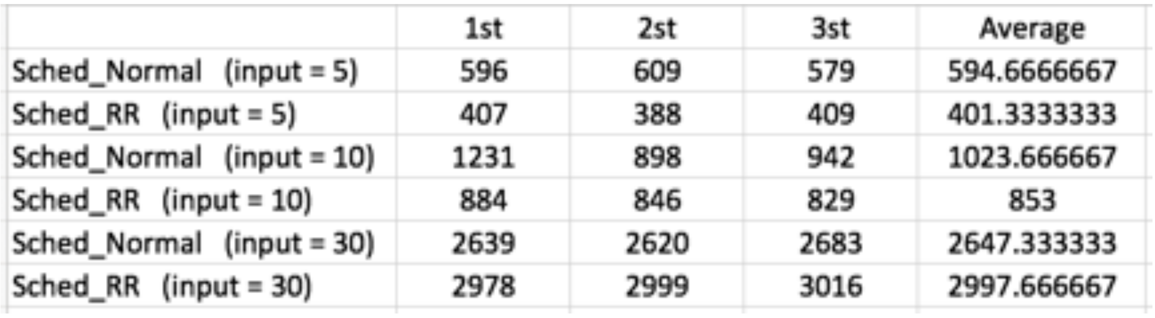


input 123456	1st	2st	3st	Average
Sched_Normal	14.455	14.822	14.455	14.57733333
Sched_FIFO	13.273	13.401	12.969	13.21433333
Sched_RR	12.928	13.313	13.215	13.152



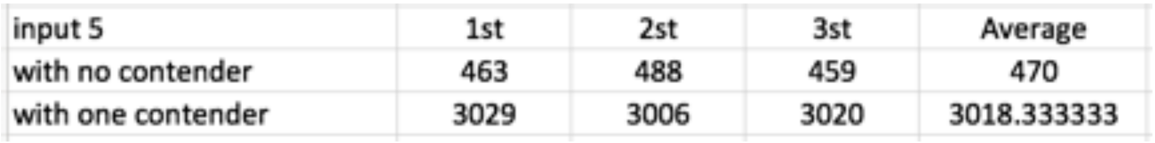
input 12345678	1st	2st	3st	Average
Sched_Normal	1562.494	1560.524	1552.073	1558.363667
Sched_FIFO	1429.571	1550.161	1349.736	1443.156
Sched_RR	1540.003	1508.942	1544.13	1531.025

c) Change the zygote's descendants' scheduler to RR, priority 10 while change the test application (T.A.) to RR, priority 99.

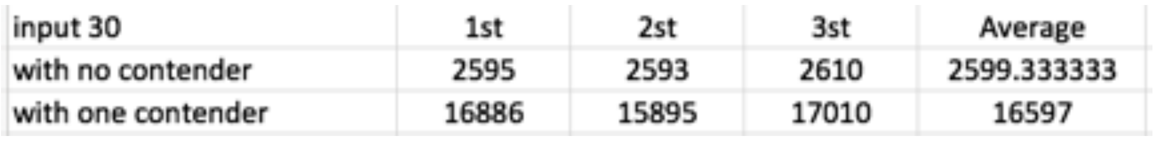


	1st	2st	3st	Average
Sched_Normal (input = 5)	596	609	579	594.6666667
Sched_RR (input = 5)	407	388	409	401.3333333
Sched_Normal (input = 10)	1231	898	942	1023.666667
Sched_RR (input = 10)	884	846	829	853
Sched_Normal (input = 30)	2639	2620	2683	2647.333333
Sched_RR (input = 30)	2978	2999	3016	2997.666667

d) Measure the time cost of T.A.'s application after modified the kernel with or without the contender. (app id = 1171)



input 5	1st	2st	3st	Average
with no contender	463	488	459	470
with one contender	3029	3006	3020	3018.333333



input 30	1st	2st	3st	Average
with no contender	2595	2593	2610	2599.333333
with one contender	16886	15895	17010	16597

2、The analysis of the OS Project 2’s result

Now, let me to analyse the result of my experiment.

In Linux, RT(real time) tasks own priority of 0 to 99 ,while normal tasks own 100 to 139. And Linux provides two kinds of RT schedulers — SCHED_FIFO and SCHED_RR.

SCHED_FIFO implements first-in-first-out algorithm without exhausting. And a runnable SCHED_FIFO task always has higher priority than any SCHED_NORMAL tasks. And only a higher priority SCHED_FIFO or SCHED_RR task can preempt a SCHED_FIFO task.

SCHED_RR is identical to the SCHED_FIFO policy to some extent. But it has its own exhausting mechanism. That is when a process ran too long, other processes (have same priority）can take place of the running process and use its CPU resources. But we should notice that the processes with lower priorities can never use the resources of higher processes.

a) The analysis of result a’s data

According to the data in the **result a**’s table. We can easily see that when the input is 5, the scheduling method SCHED_FIFO and SCHED_RR is much superior to the SCHED_NORMAL . That is because the priority of the SCHED_FIFO and SCHED_RR are much higher than SCHED_NORMAL. So the test application will be executed earlier if it is set to be FIFO or RR. Besides, because of its exhausting mechanism, the RR policy is much more efficient than FIFO, which in fact we’ve proved in class.

However, things change when the input number increases. And when the input number rises up to 30, the SCHED_FIFO and SCHED_RR policies seem to be inferior to the SCHED_NORMAL. It is weird. But considering I am using the ADB machine to run the application. When the complexity of the application rising, the context exchange costs will be rising simultaneously. And when the number comes to 30, the context exchange uses a lot of time. And the RR policy is now harmed by its exhausting mechanism, wasting more times in context exchange. In this case, the FIFO is slower than NORMAL, and RR and FIFO uses almost same amount of time.(the RR is supposed to be quicker originally).

And when compared with the condition competing with the contender program. No matter the inputting number is small or large, when the applications competing(Set to same priority), it costs longer time for one application to complete its own job. That is because the RR’s exhausting mechanism.

b) The analysis of result b's data

The b's result is about my own testing app wrote in C. And you can see the consistence when the computing complexity grows, that is the RR policy is always uses shorter time than the NORMAL policy. That is maybe because when use the program running in terminal, the context exchange doesn't happen so frequently.

c) The analysis of result c's data

The data of result c's data shows clearly the superiority of the RR policy to Normal policy. When the scheduler of descendants are set to be RR. The time usage is much smaller than the original policy — SCHED_NORMAL. And the situation happens again when number comes to 30 that the RR uses more time than NORMAL. And the reason I have already explained in the analysis (a).

d) The analysis of result d's data

The result (d) is also easy to explained.

The result (d) is in fact as same as the result in (c) for in the (d)'s experiment, I also changes the zygote's descendants' scheduler to RR. The only difference between to experiments is that the experiment (d)'s processes' RR policies are set to own different priorities, which is relative with the pids.

Now let's see the data of result (d). The above analysis has already explained the reason why the time usage become lager when the input number increases and contender program participates in disputing for CPU resources.

The interesting thing is to comparing the result (d) with result (c). Taking the inputing number is 5, without competitor's situation as instance, the result (d)'s running speed is not as quick as result (c). In fact, this is because the test app's pid is 1171. And the priority of the test app is calculated to be 21, while there are many processes' priorities calculated as 41,61,81. The low priority slows down the test app's running speed.

Ending & Thanks

This is all of my report. Thanks for the T.A. Bo Wang’s helping. I think I’ve learned a lot in this project.