OPERATING SYSTEMS

ASSIGNMENT – 3 (WRITEUP)

RAHUL KHATOLIYA

2019265

CSAM'2023

# DESCRIPTIONS OF THE CODES , IMPLEMENTATION AND ERRORS HANDLED :

1. **Changes "Linux-5.9.1/kernel/fair.c" (IMPLEMENTATION AND LOGIC):**

   Here we have made changes to two functions , which are :

   A. **" update_curr(struct cfs_rq *cfs_rq) "** in which , we are decrementing the **soft** value of a process by the **delta_exec** until the **soft** value reaches **0** or below, and as soon it reaches **0** or below, we don't go further , but provide the original definition where **vruntime** of a process is updated by incrementing it with calculated value using the method "**calc_delta_fair()**" .The reason for the change is that , since we are giving **high priority** to a process having a less **soft** value thus , we are making sure that that other similar **soft** with low priority don't undergo starvation , and the one with **high priority** remains at its urgent peak .

   B. **" *__pick_next_entity(struct sched_entity *se) "** in which we are looping the **task_struct** to find a process having least non-zero **soft** value , hence returning it to schedule next , unlike the original one where we are returning the **leftmost** node of the **rb_tree** to get process with least **vruntime** . And the reason for the change is that , In our requirements we need to schedule first the process which have **soft real time guarantee** hence , first looking the process with least non-zero **soft** value , if no such process found , then **schedule** according to the original **vruntime** functionality.

# CODE FOR CHANGED FUNCTIONS IN “ Linux-5.9.1/kernel/fair.c ” :

(**NOTE** : CHANGES ARE IN BOLD)

```
# static void update_curr(struct cfs_rq *cfs_rq)
{
        struct sched_entity *curr = cfs_rq->curr;
        u64 now = rq_clock_task(rq_of(cfs_rq));
        u64 delta_exec;
        if (unlikely(!curr))
                return;
        delta_exec = now - curr->exec_start;
        if (unlikely((s64)delta_exec <= 0))
                return;
        curr->exec_start = now;
        schedstat_set(curr->statistics.exec_max,
                     max(delta_exec, curr->statistics.exec_max));
        curr->sum_exec_runtime += delta_exec;
        schedstat_add(cfs_rq->exec_clock, delta_exec);

        if(curr->rtnice>0)
        {
                curr->rtnice = curr->rtnice-delta_exec ;
        }
        else
        {
                curr->vruntime = curr->vruntime+calc_delta_fair(delta_exec,
curr);
        }
        update_min_vruntime(cfs_rq);

        if (entity_is_task(curr)) {
```

```
        struct task_struct *curtask = task_of(curr);
        trace_sched_stat_runtime(curtask, delta_exec, curr->vruntime);
        cgroup_account_cputime(curtask, delta_exec);
        account_group_exec_runtime(curtask, delta_exec);
    }

    account_cfs_rq_runtime(cfs_rq, delta_exec);
}
```

# static struct sched_entity *__pick_next_entity(struct sched_entity *se)

```
  {
    struct rb_node *next = rb_next(&se->run_node);
    struct task_struct *task ;
    struct sched_entity *next_task ;
    unsigned long long int min_rtnice=0;
    int isfirst=0;
    for_each_process(task)
    {
        struct sched_entity *curr = &(task->se);
        unsigned long long int rtnice = curr->rtnice ;
        if(rtnice > 0)
        {
            if(isfirst==0)
            {
                min_rtnice = rtnice;
                next_task = curr;
                isfirst++;
            }
            if(rtnice<min_rtnice)
            {
                next_task = curr;
                min_rtnice=rtnice;
            }
        }
        }
        if(min_rtnice!=0)
        {
                return next_task;
        }
        if (!next)
                return NULL;
        return rb_entry(next, struct sched_entity, run_node);
    }
```

2. SYSTEM CALL CODE (rtnice.c) (IMPLEMENTATION , LOGIC AND ERROR HANDLED) :

   # HEADERS USED :

```
        #include <linux/kernel.h>
        #include <linux/init.h>
        #include <linux/sched.h>
        #include <linux/mm.h>
        #include <linux/errno.h>
        #include <linux/syscalls.h>
        #include <linux/module.h>
        #include <linux/uaccess.h>
        #include <linux/fs.h>
        #include <linux/fcntl.h>
        #include <linux/file.h>
```

   Following the Syscall Macro convention , we have to provide ‘N’ as the number of argument to be passed in the system call , and hence we have used the definition as SYSTEM_DEFINE2 , Since in our case we have to pass 2 arguments, i.e Process ID and the soft real time value .

   In first segment , we have checked whether the provided PID and the soft value is negative or not , if yes then simply return the error_numbercorresponding to the invalid argument i.e **-EINVAL** using header **<errno.h>** which helps to handle errors and throw corresponding error code .

Moreover , when used such negative values ,
using **Perror** we will print on the terminal **" rtnice : Invalid argument "** , and using **dmesg** command , we can see **" ERROR FOUND : INVALID ARGUMENT "** , which we have printed inside the **kenel** using **printk()** .

Going further , if arguments passed are **valid** , we will loop through the **task_struct** and find the process with given **pid** , and if found then we will update this process's **rtnice(soft)** value with the provided value , and if no process with this **pid** , then throw out the corresponding **error** number i.e **-ESRCH** and printing out in the kernel **" ERROR FOUND : NO PROCESS WITH THIS PID "** . In case if everything works fine , we will return 0 for invoking success to the system call.

# CODE (rtnice.c) :

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/sched.h>
#include <linux/mm.h>
#include <linux/errno.h>
#include <linux/syscalls.h>
#include <linux/module.h>
#include <linux/uaccess.h>
#include <linux/fs.h>
#include <linux/fcntl.h>
#include <linux/file.h>

SYSCALL_DEFINE2(rtnice, long, _pid, long, rtnice)
{
   int found = 0;
   if(_pid<0 || rtnice<0)// Negative PID and Soft Real time value not allowed .
   {
```

```
            printk("ERROR FOUND : INVALID ARGUMENT \n");
        return -EINVAL;
      }
      else
      {
       struct task_struct *task;
       unsigned long long soft = rtnice*1000000000;
       for_each_process(task){
        if((long)task->pid==_pid)
        {
         found = 1;
         task->se.rtnice = soft;
         printk("%llu\n",task->se.rtnice);
        }
       }
       if(found==0)
       {
        printk("ERROR FOUND : NO PROCESS WITH THIS PID \n");
        return -ESRCH;
       }
      }
          return 0;
    }
```

3. TEST CODE (test.c) :

# HEADERS USED :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <time.h>
```

```
#include <sys/time.h>
#include <ctype.h>
#include <sys/wait.h>
```

Here the Code is divided into three segments , first is **“Process selection”** , then **“Taking soft real time value from the user”** and the last **“Creating and Executing Processes”** .

In the **first segment** user will be asked to select any process among **1** and **2** , we have used **do-while loop** , so that as soon as the **user** has provided **valid** value i.e **1 or 2** the loop will terminate , otherwise it keeps on asking the valid input if user provides some irrelevant values , and to achieve the same we have made use of a helper function **check_input()** which will return **0** if the input is valid , otherwise

-1.

Now in the **second section** , again we take **user input** inside **do-while loop** but this time asking for **soft real time value** , and we are checking that user should not give **irrelevant** values like **alphanumeric or negative values** , and to make sure such cases , again we have made use of **check_input()** method explained above and a small **if-else** check for **negative value** , hence not **terminating loop** , unless the same is achieved .

In the final section , we proceed by calling the **fork() system call** , to create two **processes** , and using the earlier input of user we select to apply system call to one and ignoring other . Now to demonstrate the same we have first printed out the **process id’s** of both the processes and then **recording** the execution time of each , and printing out which terminated earlier and which later at last . Both the processes were given **identical tasks** , i.e **looping from 0 to 1999999999** and incrementing a **variable** inside the same loop . Moreover , time recording is done using the **clock_t** of **<time.h>** , where we have invoked it just above the processes’ **start** and then noting the finished time , subtracting the start time and finally dividing it by **CLOCKS_PER_SEC** to get optimal **units of time** .

CODE (test.c) :

```
/* Name: Rahul Khatoliya
   Roll_Number: 2019265 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <time.h>
#include <sys/time.h>
#include <ctype.h>
#include <sys/wait.h>

int check_input(char input[]){
	for (int i=0;input[i]!='\0'; i++){
		if(i==0){
			if(input[i]=='-'){
				continue;
			}else if(!isdigit(input[i])){
				return -1;
			}
		}
		else if(!isdigit(input[i]))
```

```
       {
           return -1;
       }
          }
          return 0;
}

int main()
{
          /*Input for Selecting Process*/
          int ch=-1;
          int check_int=-1;
          char buf[50];
          do{
                    printf("\nSelect process on which syscall should be applied : \n\n
Process 1 \n\n Process 2 \n");
                    scanf("%s",buf);
                    check_int=check_input(buf);
                    if(check_int<0){
                              printf("\nInvalid Argument");
                    }else {
                              ch=atoi(buf);
                              if(ch!=1 && ch!=2){
                                        printf("\nNo Such Process, should be 1 or 2\n");
                                        ch=-1;
          }else{
              ch=0;
          }
      }
  }while(ch==-1 || check_int==-1);

  /*Input For Soft Real Time Value*/
  int check_neg=-1;
  int check=-1;
  int rtnice;
  char soft[50];
  do{
      printf("Soft Real time Value : ");
scanf("%s",soft);
check=check_input(soft);
      if(check<0){
          printf("\nError : AlphaNumeric Not Allowed \n");
      }
      else{
          rtnice=atoi(soft);
          if(rtnice<0){
              printf("\nError : Negative Soft value not Allowed \n");
          }else{
              check_neg=0;
          }
```

```
                  }
       }while(check==-1 || check_neg==-1);

pid_t pid=fork();
if(pid!=0)
{
    printf("\nProcess 1 PID: %d\n",getpid());
    printf("\nProcess 2 PID: %d\n",pid);
    int ret;
    if(ch==1){
       ret = syscall(441,getpid(), rtnice);
       if(ret<0){
                  perror("rtnice failure");
                  exit(1);
       }
    }else{
                             ret = syscall(441,pid, rtnice);
       if(ret<0){
                  perror("rtnice failure");
                  exit(1);
       }
    }

       clock_t t;
    t = clock();
```

```
        int count=0;
           int i=0;
        for(i=0; i<2000000000; i++)
        {
            count += i;
        }
        t = clock() – t;
        double time_taken = ((double)t)/CLOCKS_PER_SEC;
        printf("\nProcess with PID: %d terminated in [ %f Time units
]\n",getpid(),time_taken);
        wait(&pid);
    }
    else
    {
           clock_t t;
        t = clock();
        int count=0;
           int i=0;
        for(i=0; i<2000000000; i++)
        {
            count += i;
        }
        t = clock() – t;
        double time_taken = ((double)t)/CLOCKS_PER_SEC;
        printf("\nProcess with PID: %d terminated in [ %f Time units
]\n",getpid(),time_taken);
    }
    return 0;
}
```