Gitansh Raj Satija OS Assignment3

Documentation

Scheduling:

CFS keeps account for the time that each process runs to ensure that each process runs only for its fair share of the processor. CFS uses the scheduler entity structure, struct sched_entity, defined in include/kernel/sched.h , to keep track of process accounting. Hence, we define rt_nice here as a u64 data type.

```
struct sched_entity {
        /* For load-balancing: */
        struct load_weight              load;
        struct rb_node                  run_node;
        struct list_head                group_node;
        unsigned int                    on_rq;

        u64                             exec_start;
        u64                             sum_exec_runtime;
        u64                             vruntime;
        u64                             rt_nice;
        u64                             prev_sum_exec_runtime;

        u64                             nr_migrations;
```

Next, we initialize the rtnice value by 0 in the sched_fork() process which is a basic setup used by init_idle() and sets up an idle thread for a given CPU. This change is done in the kernel/sched/core.c file.

```
static void __sched_fork(unsigned long clone_flags, struct task_struct *p)
{
        p->on_rq                        = 0;

        p->se.on_rq                     = 0;
        p->se.exec_start                = 0;
        p->se.sum_exec_runtime          = 0;
        p->se.prev_sum_exec_runtime     = 0;
        p->se.rt_nice                   = 0;
```

After that we make changes in two of the functions present in the kernel/sched/fair.c file, which implements the CFS scheduler.
First we make changes in the, static void update_curr(struct cfs_rq *cfs_rq) function as shown below. The update_curr function updates the current task's runtime statistics i.e. calculates the execution time of the current process and stores that value in delta_exec which is given to the vruntime. However, for defining our scheduler based on rtnice value, for all the processes which have rtnice>0, we subtract delta_exec from the rtnice value thus updating it and prioritizing.

```
/*if rtnice value is more than 0, that means process has soft real time requirements.
Therefore, for each such process we add a scheduling modification.*/
if(rtval!=0)
{
        //printk("updating rtnice value based on execution done");
        //here we subtract the time of execution and update the new rtnice value
        if(delta_exec>=rtval)
        {
                curr->rt_nice = 0;

        }
        else
        {
                curr->rt_nice = rtval - delta_exec;
        }
}
else
{
        //if process does not have soft real time requirements then process as original.
        curr->vruntime += calc_delta_fair(delta_exec, curr);
}
```

Next, we also update the entity_before function as shown below. Since, a process with higher rtnice value (soft real time requirements) implies higher priority, we return 1 or 0 based on which of the tasks to be scheduled has a higher value of rtnice. In case both have the same rtnice (or no rtnice) then the normal method using vruntime scheduling is used.

```
static inline int entity_before(struct sched_entity *a,
                                struct sched_entity *b)
{
        if(a->rt_nice>0 && b->rt_nice>0){
                printk("Two processes having rtnice compared");
                if(a->rt_nice > b->rt_nice)
                        return 1;
                else
                        return 0;
        }
        else if(a->rt_nice>0 || b->rt_nice>0)
        {
                //printk("A process having rtnice value compared with another");
                if(a->rt_nice > b->rt_nice)
                        return 1;
                else if(a->rt_nice < b->rt_nice)
                        return 0;
                //if rtnice is same then compare acc to vruntime
                else
                        return (s64)(a->vruntime - b->vruntime) < 0;
        }
        else{
                return (s64)(a->vruntime - b->vruntime) < 0;
        }
}
```

The entity_before function is called by the pick_next_entity which decides which process is to be scheduled next.

```
static struct sched_entity *
pick_next_entity(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
        struct sched_entity *left = __pick_first_entity(cfs_rq);
        struct sched_entity *se;

        /*
         * If curr is set we have to see if its left of the leftmost entity
         * still in the tree, provided there was anything in the tree at all.
         */
        if (!left || (curr && entity_before(curr, left)))
                left = curr;

        se = left; /* ideally we run the leftmost entity */

        /*
         * Avoid running the skip buddy, if running something else can
         * be done without getting too unfair.
         */
        if (cfs_rq->skip == se) {
                struct sched_entity *second;

                if (se == curr) {
                        second = __pick_first_entity(cfs_rq);
                } else {
                        second = __pick_next_entity(se);
                        if (!second || (curr && entity_before(curr, second)))
                                second = curr;
                }

                if (second && wakeup_preempt_entity(second, left) < 1)
                        se = second;
        }

        /*
```

System Call:

Following is the implementation of the rt_nice syscall, which assigns a rtnice value to a task. SYSCALL_DEFINE2, represents the fact that the system call implemented takes in 2 arguments, both of which are of long type as seen in the code below. The two arguments are pid (whose rtnice value is to be updated), and an rtnice value. Now, we define a task_struct. Then, we iterate through all the available processes using the for_each_process function and the task whose pid is equal to the pid provided as an argument to the system call is assigned the value as given by the user. The rtnice value is multiplied by 10^9 as during scheduling operations are done in nanoseconds so we need to have the rtnice value of similar type. To demonstrate the successful completion of the above said task, we find the task with the pid passed as argument using pid_task() function and then try to print its updated rtnice value in the kernel logs.

```c
SYSCALL_DEFINE2(rt_nice, long, pid, long, rtinp)
{
        struct task_struct *task;

        if(rtinp<0 || (rtinp>0 && rtinp<5))
        {
                printk("Invalid soft time value");
                return 1;
        }
        if(rtinp>100000000000)
        {
                printk("Invalid soft time value");
                return 1;
        }
        int f=1;
        for_each_process(task)
        {
                if(pid==(long)task->pid)
                {
                        f=1;
                        printk("Original rtnice value  %lld\n",task->se.rt_nice);
                        task->se.rt_nice=rtinp*1000000000;
                }
        }
        task = pid_task(find_vpid(pid), PIDTYPE_PID);
        if (task == NULL)
        {
                printk("Process with given PID not found\n");
                return 1;
        }
        printk("New rtnice value (in ns) %lld\n",task->se.rt_nice);
        if(f==0)
        {
                printk("Process with given PID not found\n");
                return 1;
        }
        return 0;
}
```

Testing:

This is the testing program. We take input the rtnice value from the user. Then, our aim is to prove the scheduling modification we have made. For this we use fork() to create a child process. Now, the child process and the parent process essentially do the same thing i.e. execute a large loop of around 10^9. However, the child function also calls the system call rt_nice, that updates the rtnice value of the task corresponding to the pid passed to the syscall. So, we provide the pid of the child process by using the getpid() call. Now to find the execution time, we use the gettimeofday() function. A common call for noting the start point and then a second call just after the execution of loop finishes in the processes. COnverting the time to milliseconds format we note and print the execution time for both the processes (by subtracting the end time from start time). Note: if the size of the loops is very less, then the processes may not get prioritized as the execution would finish within the time slice for a process and so context switching would not occur. Further note that, even if we make the function of the parent process smaller (which has less iterations and runtime) even then, the child process would be executed first as it

would be having higher rtnice value. For implementing the syscall, it was declared in the syscalls_64 table where it was assigned number 336 and entered as common (working for both 32 and 64 bit). Similarly, in the syscalls.h file, the syscall function was declared,

```c
int main() {
        long rtinp;
        struct timeval time1,time2;
        printf("Enter valid rtnice value:");
        scanf("%ld",&rtinp);
        long pid = fork();
        gettimeofday(&time1, NULL);
        if(pid < 0) {
                perror("fork error");
                exit(errno);
        }
        else if(pid != 0) {
                long i=0;
                do{

                        i++;
                }while(i<1000000000);
                gettimeofday(&time2, NULL);
                double parT= (double) (time2.tv_usec-time1.tv_usec)/1000 + (double) (time2.tv_sec-time1.tv_sec)*1000;
                printf("Execution time taken by parent = %lf\n",parT);
                wait(NULL);
        }
        else {
                long pid_update=getpid();
                int sc=syscall(336,pid_update,rtinp);
                if(sc != 0) {
                        printf("Error in syscall. Check kernel logs\n");
                }
                long i=0;
                do{
                        i++;
                }while(i<1000000000);
                gettimeofday(&time2, NULL);
                double childT= (double) (time2.tv_usec-time1.tv_usec)/1000 + (double) (time2.tv_sec-time1.tv_sec)*1000;
                printf("Execution time taken by child = %lf\n",childT);
        }
        return 0;
}
```

Diff File:

Diff -Nur command was used to make the diff.txt file, this was done after using the make distclean command which makes the file almost like the untarred version removing all the .o files and any config files if they exist.

Input and Output:

Following we have described how input/output works for the assignment. We use the make run command to compile and run the test.c file which implements our system call that assigns rtnice value provided by user to a corresponding process and returns the total time taken by the child and parent processes which essentially do the same thing however, the child process is assigned the rtnice value provided by the user.

As we can see below, we enter a valid rtnice value that will be assigned to the child process. The rtnice value must be more than equal to 5 (this is because if rtnice value is very small then it will be smaller than delta_exec and will be insignificant. Since all other priorities like vruntime which decide the scheduling process, are stored as nanoseconds, hence we need a value within that range for rtnice criteria to

work properly. As you can see below, the child process always executes first as it has a higher rtnice value (or in this case the only process to have a positive rtnice value).

```
gitsat@ubuntu:~/Desktop/Q1$ make run
gcc test.c
./a.out
Enter valid rtnice value:100
Execution time taken by child = 1776.955000
Execution time taken by parent = 3550.325000
gitsat@ubuntu:~/Desktop/Q1$ make run
gcc test.c
./a.out
Enter valid rtnice value:20
Execution time taken by child = 1901.412000
Execution time taken by parent = 3774.561000
```

Following is the message stored in kernel logs viewed by using dmesg command.

```
[  455.686849] Original rtnice value  0
[  455.686852] New rtnice value (in ns) 100000000000
[  495.089513] Original rtnice value  0
[  495.089515] New rtnice value (in ns) 20000000000
```

However, if we enter rtnice as 0(zero), we see that the child and parent take comparatively similar execution times as they are now prioritised based on the vruntime since both have the same value of rtnice (i.e. 0).

```
gitsat@ubuntu:~/Desktop/Q1$ make run
gcc test.c
./a.out
Enter valid rtnice value:0
Execution time taken by child = 3547.856000
Execution time taken by parent = 3551.776000
gitsat@ubuntu:~/Desktop/Q1$ make run
gcc test.c
./a.out
Enter valid rtnice value:0
Execution time taken by parent = 3553.974000
Execution time taken by child = 3553.869000
```

.Error Handling

Following are bits and pieces of codes which handle errors. These are defined both in the syscall implementation as well as in the test.c file. We use functionalities like perror to receive error messages corresponding to an error number. Further, we return 1 in case of errors and 0 in case of success.

```
if(pid < 0) {
        perror("fork error");
        exit(errno);
}
```

```
 if(rtinp<0 || (rtinp>0 && rtinp<5))
 {
         printk("Invalid soft time value");
         return 1;
 }
```

```
if (task == NULL)
{
        printk("Process with given PID not found\n");
        return 1;
}
```

```
if(f==0)
{
        printk("Process with given PID not found\n");
        return 1;
}
```

Below we have shown the error handling part of the assignment. If a user enters an invalid pid like a negative entry(or out of range value i.e. very high (>10^11)) Then the syscall returns with an error, a message corresponding to which is stored in the kernel logs.

```
gitsat@ubuntu:~/Desktop/Q1$ make run
gcc test.c
./a.out
Enter valid rtnice value:-10
Error in syscall. Check kernel logs
Execution time taken by parent = 3567.352000
Execution time taken by child = 3562.725000
```

If in case the pid given to the system call is invalid i.e. the pid doesn't correspond to a non null process, then also error will be shown in the system call.

```
gitsat@ubuntu:~/Desktop/Q1$ make run
gcc test.c
./a.out
Enter valid rtnice value:23
Error in syscall. Check kernel logs
```

```
[  583.209861] Invalid soft time value
[ 1055.319332] Process with given PID not found
```

Some errors along with their codes are mentioned below

1 /* Operation not permitted */

```
3          /* No such process */
4          /* Interrupted system call */
10         /* No child processes */
12         /* Out of memory *
32        /* Broken pipe */
```