

# CS 411 Project 1 Group Writeup

Doug Dziggel

Ke Fan

David Merrick

Michael Phan

April 15, 2013

## **Abstract**

This paper describes the design process and implementation of the FIFO and round-robin scheduling algorithms in the Linux kernel.

# 1 Design

We based our design for round-robin and FIFO scheduling policies on the actual implementation of them in the Linux 3.0.4 kernel. This allowed us to leverage the existing functions, definitions, and data structures that were already written in the kernel provided for us for the project. In several respects, round-robin and FIFO policies work very similarly and call many of the same functions. Round-robin defines a timeslice for each process to run, then runnable processes that are set to use this policy are placed in a runqueue. Each process executes for the amount of time specified by the timeslice definition, then the next process in the queue is run. This cycle repeats itself, running each of the processes in the queue in turn for an equal amount of time. If there are no other runnable SCHED\_RR processes in the queue, the current process continues to run. SCHED\_FIFO is implemented the same way, but with infinite timeslices. This causes each process to run until it either completes or blocks and yields the CPU. When the CPU is free from that process, the next runnable SCHED\_FIFO task in the runqueue begins execution.

## 2 Implementation

System calls/functions we edited:

**In sched.c:**

1. static inline int rt\_policy(int policy)

We added “if (unlikely(policy == SCHED\_FIFO || policy == SCHED\_RR)) return 1;” to this function. The unlikely() function is a compiler optimization to tell the compiler to favor the more likely side of a jump instruction. It is essentially a hint that tells the compiler which direction the logic is likely to go. The code added to rt\_policy function causes it to return 1 if the task’s policy is either SCHED\_RR or SCHED\_FIFO. Other options for policy are SCHED\_BATCH, and SCHED\_IDLE, and SCHED\_NORMAL, which is the most common way to schedule processes (it is better known as the Linux Completely Fair Scheduler, CFS). The point of rt\_policy is to return true (1) if a task has a real-time policy (SCHED\_RR or SCHED\_FIFO) and false (0) otherwise. Note: Since SCHED\_NORMAL is not a real-time policy, it is more likely that the policy is non-realtime so it makes sense for the unlikely() compiler optimization to be here.

rt\_policy() is called in the task\_has\_rt\_policy function, which is primarily used in other functions throughout sched.c to determine how to set the priority of a task. So without the rt\_policy function returning 1 in the event of a realtime policy, none of the other functions set the task priority accordingly and this breaks both the SCHED\_FIFO and SCHED\_RR algorithms.

2. void sched\_fork(struct task\_struct \*p)

Inside the if (unlikely(p->sched\_reset\_on\_fork))... if statement, we added the following to occur first:

```
if (p->policy == SCHED_FIFO || p->policy == SCHED_RR) {
    p->policy = SCHED_NORMAL;
    p->normal_prio = p->static_prio;
}
```

These lines of code will check if the current task policy is either FIFO or RR. If so, it will set the scheduling policy back to normal and set the normal priority of the task to become static. It allows the task to calculate the nice value, time slices, interactivity, and dynamic priority.

This function also resets the schedule policy of the child in the event this is specified in the parent.

3. static int \_\_sched\_setscheduler(struct task\_struct \*p, int policy, const struct sched\_param \*param, bool user)

In the given kernel files, the \_\_sched\_setscheduler function was missing some statements for FIFO and RR in the if statement:

```
if (policy != policy != SCHED_NORMAL && policy != SCHED_BATCH &&
    policy != SCHED_IDLE)
    return -EINVAL;
```

We replaced the statement to include policies for FIFO and RR:

```
if (policy != SCHED_FIFO && policy != SCHED_RR &&
    policy != SCHED_NORMAL && policy != SCHED_BATCH &&
    policy != SCHED_IDLE)
    return -EINVAL;
```

4. SYSCALL\_DEFINE1(sched\_get\_priority\_max, int, policy)

Inside this function, there is a switch case statement for the current scheduling policy. We added a few lines of code to include cases for FIFO and RR:

```
case SCHED_FIFO:
case SCHED_RR:
    ret = MAX_USER_RT_PRIO-1;
    break;
```

This function returns the maximum priority value for a scheduling policy. By including these two cases, the function sched\_get\_priority\_max will return the value MAX\_USER\_RT\_PRIO-1 for the scheduling policies FIFO and RR. In the case of SCHED\_FIFO and SCHED\_RR, these priorities can be between 1 and 99. Since MAX\_USER\_RT\_PRIO is defined as 100, the maximum priority value this function will return for SCHED\_RR and SCHED\_FIFO policies is 99.

5. SYSCALL\_DEFINE1(sched\_get\_priority\_min, int, policy)

```
case SCHED_FIFO:
case SCHED_RR:
    ret = 1;
    break;
```

This function returns the minimum priority value for a scheduling policy. In the case of SCHED\_FIFO and SCHED\_RR, this is 1.

#### In sched\_rt.c:

6. static void task\_tick\_rt(struct rq \*rq, struct task\_struct \*p, int queued) Inside this function, there is a block of comment with nothing following it. It states:

```
/*
 * RR tasks need a special form of timeslice management.
 * FIFO tasks have no timeslices.
 */
```

The following lines of code were added after that comment:

```
if (p->policy != SCHED_RR)
    return;

if (--p->rt.time_slice)
    return;

p->rt.time_slice = DEF_TIMESLICE;
```

This begins by checking if the current policy is RR. If it is not, then it will return from the function to end. If the previous task still has a time slice, it will return from the function to end. If neither of the previous cases occur, then it will continue to set the current task to have a time slice equal to DEF\_TIMESLICE.

7. static unsigned int get\_rr\_interval\_rt(struct rq \*rq, struct task\_struct \*ttask)

This function's purpose is to get the RR interval time. In the kernel provided for us for the project, it initially would always return 0, which would cause RR to become FIFO. The following lines of code were added before the return 0 to ensure that when the policy is RR, the function correctly returns the timeslice value.

```
if (task->policy == SCHED_RR)
    return DEF_TIMESLICE;
else
{
```

## 3 Testing Code

```
#include <unistd.h>
#include <sched.h>
```

```

#include <time.h>
#include <stdio.h>
#include <stdlib.h>

//Defines the policy to use for this task for testing
#define PROCESS_POLICY SCHED_RR          //Change to SCHED_FIFO to test FIFO scheduling
#define TIMESLICE 0.1

//Struct needed for sched_setscheduler() to function correctly:
struct sched_param param;

void
main()
{
    int i;
    int j;

    param.sched_priority = sched_get_priority_max(PROCESS_POLICY);
    //A "0" as the first parameter to sched_setscheduler() means
    //set policy for this process
    if (sched_setscheduler(0, PROCESS_POLICY, &param) == -1) {
        printf("sched_setscheduler broke\n");
        exit(-1);
    }

    unsigned long mask = 8;          /* processors 4 */
    unsigned int len = sizeof (mask);
    //Set the CPU affinity to CPU 0
    if (sched_setaffinity(0, len, &mask) < 0) {
        printf("sched_setaffinity not working boss \n");
        exit(-1);
    }

    pid_t pid;
    clock_t start, stop, start1, stop1;
    printf("START\n");
    double time_elapsed, print_time;
    //Fork off four child processes
    for (i = 0; i < 4; i++) {
        switch (pid = fork()) {
            case -1:          //oops case
                exit(-1);
            case 0:          //child case
                j = 0;
                while (j < 4) {
                    //Time the print operation
                    start = clock();
                    printf("Parent: %d PID: %d Iter: %d\n", i,
                        getpid(), j);
                    stop = clock();
                    print_time =

```

```

        (double) (stop - start) / CLOCKS_PER_SEC;

start = clock();
time_elapsed = 0;
//Busy-wait until the end of the timeslice
while ((time_elapsed + print_time) < TIMESLICE) {
    asm("");
    /* ^Assembly language code added to
    * prevent compiler from potentially
    * optimizing out the following lines
    */
    stop = clock();
    time_elapsed =
        (double) (stop -
                start) / CLOCKS_PER_SEC;
}
printf("TIMESLICE %f\n",
        (time_elapsed + print_time));
j++;
}
_exit(EXIT_SUCCESS);
default:    //parent case
    break;

    }
}
for (i = 0; i < 4; i++) {
    //Reap the souls of our dead children
    wait();
}
printf("finished\n");
}

```