1 Introduction

This report summarizes our implementation of Linux Kernel Programming Project 3. In this report we describe the well-known Not Completely Fair Scheduler (§2), discuss our implementation (§3), and show our evaluation of correctness (§4) and scalability (§5).

2 The Not Completely Fair Scheduler

Our Not Completely Fair Scheduler (NCFS) combines the fairness of round-robin scheduling with a priority scheme that tries to improve the overall performance of the system’s CPU-bound and I/O-bound tasks. The key idea of the NCFS is that whether a task is CPU-bound or I/O-bound may change over time, and it uses a feedback mechanism to change the priorities of tasks throughout their life. Tasks may limit the degree to which their priority can change.

We implemented a simple version of the NCFS algorithm.

Tasks set their priority level, choosing from 1 to $NUM_TIERS$. Our NCFS maps priority levels to tiers. Within each tier, we maintain a hierarchy of ranks, 1 to $NUM_RANKS$. We describe a task’s priority with a 2-tuple $< tier, rank >$. Priority is determined with a two-level sorting scheme. Tasks are first prioritized based on tier, and in the event of a tie, based on rank. The larger a task’s tier and rank values, the higher its overall priority.

The algorithm follows these basic scheduling rules.

1. **Honor coarse granularity** Tasks with higher tier values will be scheduled before tasks with lower tier values.

2. **Improve throughput** Among tasks with the same tier value, tasks with higher rank values will be scheduled before tasks with lower rank values.

3. **Be fair** Tasks with the same priority (identical $< tier, rank >$) are scheduled in a round-robin order using a fixed timeslice.

The effect is that users can differentiate at a coarse granularity between “more important” and “less important” tasks by defining the tier. Our NCFS is free to modify the rank to improve the system throughput.

The feedback rules that modify a task’s rank are modeled on the Linux CFS, and try to increase the total throughput of the system by keeping both the processor and I/O devices busy. Note that to honor the user’s coarse importance (tier), tasks never change tier, just rank.

4. **Reward I/O-bound tasks** When a task blocks for I/O before consuming its entire timeslice, its rank is increased, bounded by a maximum value of $NUM_RANKS$. 
5. **Punish CPU-bound tasks** When a task consumes its entire timeslice, it is preempted and its *rank* is decreased, bounded by a minimum value of 1.

Figure 1 illustrates the NCFS.

Unlike the CFS, we maintain no notion of a task’s history. *Rank* changes are made based solely on its behavior during its most recent timeslice. We believe that this is better than simply using the *tier* for scheduling decisions, as we suspect tasks exhibit locality in whether they are CPU-bound or I/O-bound.

One risk is that tasks may *alternate* between being CPU-bound and I/O-bound depending on the definition of the timeslice. When a task blocks for I/O, it will presumably do something with the resulting data. If its processing takes longer than the timeslice, it may be inappropriately demoted to the more CPU-bound ranks as a result. Though it will be promoted towards the more I/O-bound ranks the next time it blocks for I/O, it is for this reason that the “Nearly” is included in our scheduling algorithm’s title.

![Figure 1: NCFS Scheduling Algorithm. Each tier is the same as the one that is illustrated.](image)

**3 NCFS Implementation**

We discuss our implementation from two perspectives: data structures (§3.1) and algorithms (§3.2).

**3.1 Data Structures**

The key data structures of our NCFS are `struct ncfs_task`, `struct ncfs_tier`, and `struct ncfs_rq`. These are listed in Figure 2. All NCFS tasks live in the `struct ncfs_rq`’s *all_tasks* list. When they are runnable, they are moved to the appropriate *tier*. Each `struct ncfs_tier` maintains an array of *rank* lists into which we place the runnable NCFS task. Each rank list is a round-robin queue, with new tasks put at the tail and the next task at the head. We also maintain counts of the number of tasks in each *tier* and *rank* for debugging purposes.
/* Lives in ncfs_rq in one of the ncfs_tier's. */
struct ncfs_task {
    struct task_struct * task;
    struct list_head all_elem; /* In ncfs_rq's all_tasks. */
    struct list_head runnable_elem; /* In one of ncfs_rq's tiers. */
};

/* One of the NCFS_NUM_TIERS tiers in ncfs_rq. */
struct ncfs_tier {
    int tier_num;
    /* Runnable ncfs_task's in this tier, divided into prioritized ranks.
     * ranks[0] is the highest rank, for more I/O-bound tasks.
     * ranks[NCFS_NUM_RANKS-1] is the lowest rank, for more CPU-bound tasks.
     * ncfs_task's change ranks when timeslice expires (more CPU-bound)
     * or when they are dequeue'd without using timeslice (more I/O-bound).
     * Updated in {en,de}queue_task_ncfs.
     * Elements are ncfs_task.runnable_elem's. */
    struct list_head ranks[NCFS_NUM_RANKS];
    unsigned int rank_sizes[NCFS_NUM_RANKS]; /* Length of each list. */
};

/* Track all NCFS tasks.
 * All tasks: live in the unrunnable_tasks list.
 * Runnable tasks: 2-level hierarchy: they live in one {tier, rank} combination.
 * Invariants:
 * 1. All potentially-runnable NCFS tasks are in all_tasks.
 * 2. All runnable NCFS tasks are in a tier at the end of each hook.
 * NCFS task lifecycle:
 * 1. NCFS tasks are added to all_tasks in _ncfs_register_task.
 * 2. Once enqueue'd, they are placed in the tail of the appropriate tier.
 * 3. When dequeue'd, they are removed from the tier.
 * 4. When rank changes, they are relocated within their tier, placed at the tail of the new list.
 * They will change rank within tier based on whether they seem more CPU-bound or more I/O-bound.
 * 5. NCFS tasks are cleaned up in dequeue when the current task is dying/dead/zombie/etc.
 */
struct ncfs_rq {
    /* All NCFS tasks.
     * Elements are ncfs_task.all_elem's. */
    struct list_head all_tasks;
    /* ncfs_tiers[0] is the highest tier, for higher priority tasks.
     * Elements are ncfs_task.runnable_elem. */
    struct ncfs_tier tiers[NCFS_NUM_TIERS];
    unsigned int tier_sizes[NCFS_NUM_TIERS]; /* Size of each tier. */
};

Figure 2: Data structures of NCFS, found in kernel/sched.c.
3.2 Algorithms

The Linux scheduling subsystem consists of an underlying scheduling framework into which you can plug different schedulers. A scheduler is defined by a set of “hooks” described in the `struct sched_class`. The use of each hook is rather hazy, and we performed a careful study of the circumstances under which each hook is invoked. We were helped in this endeavor by `Documentation/scheduler/sched-design-CFS.txt` and Ishkov’s master’s thesis [1]. Our findings are in the `Documentation/scheduler/sched-hooks.txt` file in our patch.

Using this study, we identified the relevant hooks: `enqueue_task`, `dequeue_task`, `yield_task`, `check_preempt_curr`, `pick_next_task`, `set_curr_task`, `task_tick`, `task_fork`, and `get_rr_interval`.

- `enqueue_task` is straightforward: we place the runnable task in the appropriate tier, at the tail of the appropriate rank list.
- `dequeue_task` is similar: we remove the no-longer-runnable task, freeing the associated `struct ncfs_task` if it is dying. **Feedback**: In this hook we reward tasks for giving up the CPU by increasing their priority (changing their rank).
- `yield_task`: **Feedback**: we reward tasks for giving up the CPU.
- `check_preempt_curr` allows an NCFS task to preempt another NCFS task if the new task has a higher overall priority (`< tier, rank >`).
- `pick_next_task` walks down the tiers, walking down the ranks within each tier. We pick the head of the first non-empty rank we find in this search. We also set the NCFS start time in the chosen task for use in `task_tick` later on.
- `set_curr_task` sets the NCFS start time in the chosen task for use in `task_tick` later on.
- `task_tick` compares the current task’s start time against the NCFS TIMESLICE (10ms). If it has exceeded its slice, we call the scheduling framework’s `resched_task`. **Feedback**: We also punish the task for using up its entire timeslice by decreasing its priority (changing its rank).
- `task_fork` adds the new task to the `ncfs_rq`.
- `get_rr_interval` returns the NCFS’s TIMESLICE.

We also add new NCFS tasks to the `rq`’s `ncfs_rq` member during the scheduling framework’s `sched_setscheduler` method.

4 Correctness

We used white-box testing to evaluate the correctness of our implementation.

4.1 Description

The best way to test our NCFS scheduler’s correctness is to produce an external model and to compare the implementation to the model (see discussion in Appendix ??). Because of the given time, we opted for simpler test cases that verify our scheduler’s correctness. The purpose of these test cases is to verify whether our scheduler implements the NCFS rules correctly: **honor coarse granularity**, **improve throughput**, **be fair**, **reward I/O-bound tasks**, and **punish CPU-bound tasks** (as described in section 2).

We provided 9 test cases:
• Test 1 creates one I/O-bound task with specific tier and rank.
• Test 2 creates one CPU-bound task with specific tier and rank.
• Test 3 creates two I/O-bound tasks with same tiers and ranks.
• Test 4 creates two I/O-bound tasks with different tiers but same ranks.
• Test 5 creates two CPU-bound tasks with same tiers and ranks.
• Test 6 creates two CPU-bound tasks with different tiers but same ranks.
• Test 7 creates one I/O-bound task and one CPU-bound task with same tiers and ranks.
• Test 8 create one I/O-bound task with lower tier and one CPU-bound task with higher tier.
• Test 9 create one I/O-bound task with higher tier and one CPU-bound task with lower tier.

Here we explain the purpose of each test case:
• Test 1 and test 2 are used to verify whether the **reward I/O-bound tasks** rule and the **punish CPU-bound tasks** rule are implemented correctly.
• Test 3 and test 5 are used to verify whether the **be fair** rule, the **reward I/O-bound tasks** rule and the **punish CPU-bound tasks** rule are implemented correctly.
• Test 4, test 6, test 8 and test 9 are used to verify whether the **honor coarse granularity** rule, the **reward I/O-bound tasks** rule and the **punish CPU-bound tasks** rule are implemented correctly.
• Test 7 is used to verify whether the **improve throughput** rule, the **reward I/O-bound tasks** rule and the **punish CPU-bound tasks** rule are implemented correctly.

The CPU-bound task is implemented by forking a thread doing arithmetic on **volatile int**s. The I/O-bound task is implemented by forking a thread running a series of **usleeps** for random periods of time. The periods were randomly chosen to be a mix of shorter and longer than the NCFS TIMESLICE.

### 4.2 Results

Each test case passed. The output behavior of each test case is described below:

• **Test 1**: The tier of the I/O-bound task stays same, while the rank of I/O-bound task increases every task.tick. This is expected as the rule **reward I/O-bound tasks**.

• **Test 2**: The tier of the CPU-bound task stays same, while the rank of CPU-bound task decreases every task.tick. This is expected as the **punish CPU-bound tasks** rule.

• **Test 3**: The two I/O-bound tasks play “follow-the-leader”. The first sleeps, is dequeued, and rewarded. While it sleeps, the second sleeps, is dequeued, and rewarded. If the first task happens to wake up before the second task has a chance to sleep, it will preempt it, “skipping ahead”. Once both tasks have reached the highest rank, they take turns equally. This is the expected behavior of the **reward I/O-bound tasks** rule and the **be fair** rule.
• **Test 4**: The ranks of these two I/O-bound both increases every task_tick. However they have different tiers. The lower tier task can only execute when the higher tier task sleeps. Once the higher tier task wakes up, the lower tier task is preempted. This is the expected behavior of the **reward I/O-bound tasks** rule and the **honor coarse granularity** rule.

• **Test 5**: The two CPU-bound tasks play “follow-the-leader”. The first runs out of time slice, and is punished. While it is punished, the second runs out of time slice, and is punished. Once both tasks have reached the lowest rank, they take turns equally. This is the expected behavior of the **punish CPU-bound tasks** rule and the **be fair** rule.

• **Test 6**: The ranks of these two CPU-bound task both decreases every task_tick. However they have different constant tiers. The lower tier task can only execute after the higher tier task exits. This is the expected behavior of the **punish CPU-bound tasks** rule and the **honor coarse granularity** rule.

• **Test 7**: The I/O-bound task and CPU-bound task start from same tier and rank. The rank of the I/O-bound task increases every task_tick, while the rank of the CPU-bound task decreases every task_tick. Once the rank of the I/O-bound task is higher than the rank of the CPU-bound task, the CPU-bound task can only execute when the I/O-bound task sleeps. (Once the I/O-bound task wakes up, the CPU-bound task is preempted.) This is the expected behavior of the the **reward I/O-bound tasks** rule, the **punish CPU-bound tasks** rule, improve throughput rule.

• **Test 8**: The I/O-bound task has a lower tier than the CPU-bound task. The rank of the I/O-bound task increases every task_tick, while the rank of the CPU-bound task decreases every task_tick. The I/O-bound task can only execute after the CPU-bound task exits. This is the expected behavior of the the **reward I/O-bound tasks** rule, the **punish CPU-bound tasks** rule, **honor coarse granularity** rule.

• **Test 9**: The I/O-bound task has a higher tier than CPU-bound task. The rank of the I/O-bound task increases every task_tick, while the rank of the CPU-bound task decreases every task_tick. The CPU-bound task can only execute only when the I/O-bound task sleeps. Once the I/O-bound task wakes up, the CPU-bound task is preempted. This is the expected behavior of the the **reward I/O-bound tasks** rule, the **punish CPU-bound tasks** rule, **honor coarse granularity** rule.

5 **Scalability**

We took three approaches to evaluate the scalability of NCFS: stress testing (§5.1), complexity analysis (§5.2), and performance comparison (§5.3).

5.1 **Stress testing**

First, we did a “stress test” to put our scheduler through its paces at scale. Then we The suite is described in Table 1. As in §4, our CPU-bound processes do arithmetic on volatile ints, while our I/O-bound processes make a series of usleep calls. Both types check the time periodically to determine when to finish. We ran the suite with a range of parameters, using between 1 and 100 processes in total, ideal running times ranging from 1 to 5 seconds, and with tasks in one or more NCFS tiers.

The NCFS scheduler passes the suite if the kernel does not crash and if all of the processes successfully complete. Note that the set of system calls exercised by our suite is fairly limited:
### Test Suite Description and Parameters

<table>
<thead>
<tr>
<th>Test name</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpuSched</td>
<td>CPU-bound processes (arithmetic on volatile ints).</td>
<td>Number of processes, running time, distribution of priorities.</td>
</tr>
<tr>
<td>ioSched</td>
<td>I/O-bound processes (they usleep).</td>
<td>Number of processes, running time, distribution of priorities.</td>
</tr>
<tr>
<td>allSched</td>
<td>Mix of CPU-bound and I/O-bound processes.</td>
<td>Number of processes of each type, running time, distribution of priorities.</td>
</tr>
<tr>
<td>hydra</td>
<td>Like allSched, but each child may also fork itself.</td>
<td>Number of processes of each type, running time, distribution of priorities, maximum fork depth.</td>
</tr>
</tbody>
</table>

Table 1: Black-box test suite for the NCFS scheduler. Test suite is implemented in C.

**sched_setscheduler, fork, sleep, usleep, stat.** Our implementation is a proof of concept, and we suspect that using untested scheduler system calls might break our implementation.

**Results** Our NCFS implementation passes the black-box test suite on all parameters we tested. We ran the suite inside QEMU with a gdb client attached but no breakpoints enabled.

**Lessons learned**

1. **printk** is expensive. The test suite would take minutes to run with all of our `printk`s enabled, but `echo 6 > /proc/sys/kernel/printk` (nothing at or louder than KERN_INFO) reduced the test suite running time to under one minute.

2. We had to tweak the test suite a bit to address an outcome of making NCFS the highest priority scheduler. The parent forks children, each of which waits for the parent’s signal before all proceed with their assigned role. The parent sends a signal after `forking` all of the children. In the process of `forking` many children, the parent is labeled as a CPU-bound process, while the already-created children become I/O-bound because they sleep waiting for the parent to create a file to signal the next phase. When there are many children, the odds are good that there will always be some child awake and testing for the signal file, and so the children (I/O-bound – higher priority) starve the parent.

   We solved this issue by increasing the child poll interval from 20ms to 3 seconds, leaving the parent plenty of space to finish `forking` and then create the signal file. However, the more general problem illustrated by this experience is that our feedback mechanism is too blunt and aggressive. Thus, our scheduler lives up to its name: Not Completely Fair Scheduler.

### 5.2 Complexity Analysis

NCFS’s fundamentally round-robin design allows $O(1)$-time scheduling decisions regardless of the number of NCFS tasks. NCFS has $O(N)$ storage complexity, because it adds one element to its linked lists for each NCFS task.

As a result, NCFS should be low-overhead and scalable.

### 5.3 Performance Comparison

Our stress testing ran for an approximately fixed amount of time. Each child has a time limit and checks the system clock to determine when to stop.

To evaluate performance, we wanted to measure the difference in time required to compute a fixed-size task. We developed a program called `fixed_size` to this end. It behaves deterministically, forking children who do a mix of CPU-bound and I/O-bound activities in fixed order. By varying the number of children and whether or not they use NCFS, we can measure the relationship between the number of children and the amount of time required to perform the `fixed_size` tasks in NCFS vs. CFS.
The results, illustrated in Figure 3, are mixed. First, we note that our complexity analysis (§5.2) was correct: the running time of fixed_size varied linearly with the number of processes under both NCFS and CFS. Second, it appears that our NCFS is rather less efficient than CFS. We suspect that this difference is due to the liberal use of printk and if(DEBUG) statements in our NCFS implementation.

![Relative performance of tool on CFS and NCFS](image)

Figure 3: Performance of NCFS vs. CFS. Run in QEMU with attached gdb client but no breakpoints. Three runs for each number of children.

### 6 Conclusion

This was a challenging project for us. We learned a lot about debugging in the Linux kernel, although we found VirtualBox+QEMU extremely slow to work with. Next time we’ll use KVM+QEMU instead of VirtualBox+QEMU! On the other hand, our reluctance to use the debugger encouraged us to review our code and use pair programming, which we think was helpful for our correct result.

### References