

FSVP Project : Scheduler Verification

Prasad Joshi(07305047)

Jagadish M(07305050)

December 2, 2007

1 Introduction

The part of the source code we considered for verification, was the scheduler code for linux kernel “/kernel/sched.c” We tried to verify some interesting properties of the different scheduler functions.

Linux scheduling is based on the time-sharing technique. Several processes are allowed to run “concurrently,” which means that the CPU time is roughly divided into “slices,” one for each runnable process. Of course, a single processor can run only one process at any given instant. If a currently running process is not terminated when its time slice or quantum expires, a process switch may take place. Time-sharing relies on timer interrupts and is thus transparent to processes.

There are two types of processes:

Non-real time processes for which the average delay must fall between 50 and 150 ms. The variance of such delay must also be bounded, or the user will find the system to be erratic. Typical interactive programs are command shells, text editors, and graphical applications.

Real-time processes which have strong scheduling requirements. Such processes should never be blocked by lower-priority processes, they should have a short response time and response time should have minimum variance. Typical real-time programs are video and sound applications, robot controllers, etc.

Priority : The static priority is assigned when a process is created and remains unchanged during execution, unless it is modified using the Nice system call. The static priority thus expresses a process importance relative to others on the system on a longer term basis. The dynamic priority is modified as the process executes to reflect its history. The scheduler uses the dynamic value when deciding which process to select from the ready list.

2 Properties of Scheduler Verified

2.1 Property 1: Updating Dynamic Priority

When static priority of a process is changed by the user through the nice system call. The dynamic priority should get updated accordingly. More specifically, if SP_{old} and DP_{old} are the old static and dynamic priorities of a process and SP_{new} is the updated static priority then the new dynamic priority should be set to

$$DP_{new} = SP_{new} + DP_{old} - SP_{old}$$

In the `sched.c` , `set_user_nice()` is a function which can change the static priority of a process given by `p->static_prio`. Dynamic priority and can be accessed by `p->prio` where `p` is a pointer to `task_struct` which defines the general structure of a process.

`set_user_nice()` is generally called by other functions and may also be run multiple times in a loop. We have modeled this in our code considering the difference between the two priorities as the loop invariant.

The function does not behave as expected, BLAST shows the program[`dynamic_bug.c`] to be unsafe. A simple correction[`dynamic.c`] removes this error—again verified using BLAST.

2.1.1 Implementation details

The code almost resembles the actual `set_user_nice()`. We tried to resolve atleast upto 3 levels but it was very difficult to resolve all dependencies. However all macros and functions in `set_user_nice()` have been maintained either by keeping all the appropriate definitions as in original code or by adding appropriate stubs at places. The one major change is regarding passing arguments of the function. We tried using the `-talias` option ,but as it did not work ,we made the relevent parameters global. Note the use of boolean variable 'flag'. The use of flag, enables us to say that only in case of non real time process the post condition must hold. This is in accordance to what we had been instructed. To compare code , the following link may be useful.

Link: <http://lxr.linux.no/source/kernel/sched.c?v=2.6.11#L3227>

2.2 Property 2: Higher priority for Real-time process

A real time process has a goodness value greater than a non real time process.

As mentioned earlier, real-time processes have strict scheduling requirements than non-real time processes. The main function of the scheduler is to determine the next task to be run. `schedule()` function searches the runqueue (a list structure which contains all the runnable processes) and picks the processes with maximum goodness value Fig 1. The `goodness` function gives the goodness value which indicates how desirable a process is for scheduling.

If the `SCHED_OTHER` flag is non zero then the process is a non real time process. Here the goodness value of a non real time process is a function of its

- “nice” value
- current weight
- counter(number of clock ticks remaining before forceful rescheduling)

In case of real time process, a value of 1000 is added to the current `rt_priority` value. Thus by tracking the predicates that give us an idea of the bounds of weight, counter and nice values we verify that a real time process has a higher goodness value than any non real time process.

2.2.1 Implementation details

The link for comparing the code

<http://lxr.linux.no/linux-old+v2.4.31/kernel/sched.c#L144>

```

list_for_each(tmp, &runqueue(this_cpu)) {
    p = list_entry(tmp, struct task_struct, run_list);
    if (local_can_schedule(p)) {
        int weight = goodness(p, prev->active_mm);
        if (weight > c) {
            ...
            c = weight;
            next = p;
        } else if (weight > prev_next_weight) {
            prev_next_weight = weight;
            prev_next = p;
        }
    }
}

```

Figure 1: Scheduler picks the task with highest goodness value

```

p->mm=this_mm;

if ( p->counter >= 0 && p->counter < 40 )
    if (p->nice >= -20 && p->nice <= 20) {
        goodness_value1=goodness();
        p->mm=this_mm+1;
        goodness_value2=goodness();
    }

```

Figure 2: Current Memory mapped process get higher priority

2.3 Property 3: Higher priority for Memory mapped processes

Given two non real-time processes the one which has its memory structure equal to current memory mapped region gets higher priority, other variables remaining equal

We check this property in the following way:

- Set the following precondition: $0 \geq counter \leq 40 \&\& -20 \geq nice \leq 20$
- Run the goodness() function with process memory structure $p- > mm$ mapped to **this_mm**. Store the return value in value1.
- Run the goodness() function again with $p- > mm \neq this_mm$. Store the return value in value2.
- Assert that value1 is greater than value2.

The code snippet is shown in Fig 2.

3 Usage

The directory **source_code** contains three subdirectories corresponding to each property. Each of them contains the source program(.c file) the predicate file (.pred) and the script to verify

programs using BLAST. Make sure you have pblast.opt and Simplify program directories added to the PATH variable. You may verify each property by typing:

```
./run.sh
```

in each folder.

4 Conclusion

This exercise in verification turned out to be a great learning as well as enjoyable experience for both of us. We started out ambitiously with the intention of proving many interesting properties about the scheduler, but had to settle for somewhat reasonably simple properties. The task of choosing which properties to prove was as hard as verifying them—the more interesting a property was the harder it was to verify! As far as the tools go, we spent a good deal of time learning to use BLAST. We did not explore other tools like ARMC and Dagger partly because BLAST was sufficient to verify our properties and partly because of scanty documentation available on the web on these tools. One observation we made during the exercise is the amount of assumptions one needs to make before the process of verification. A couple of simple properties that we intended to check required us to simplify the code to the extent that its correspondence to the original code became doubtful! We wonder if this is the case in the “real world” too and if there any is methodical(automatable) way of resolving dependencies given a property to be proved.