Testing task schedulers on Linux system

Leonardo Jelenković, Stjepan Groš, Domagoj Jakobović
University of Zagreb, Croatia
Faculty of Electrical Engineering and Computing

***Abstract* – Testing task schedulers on Linux operating system proves to be a challenging task. There are two main problems. The first one is to identify which properties of the scheduler to test. The second problem is how to perform it, e.g., which API to use that is sufficiently precise and in the same time supported on most platforms. This paper discusses the problems in realizing test framework for testing task schedulers and presents one potential solution. Observed behavior of the scheduler is the one used for "normal" task scheduling (SCHED_OTHER), unlike one used for real-time tasks (SCHED_FIFO, SCHED_RR).**

I. Introduction

Starting from Linux kernel version 2.6.23 task scheduler was changed from *O(1)* [1] to *Completely Fair Scheduler* (CFS) [2]. This caused a lot of discussion in the Linux community [3]. Arguments used to support the claim that "*this* scheduler is better than *the other*" varies from (subjective) user experience in everyday activities to (objective) results of some benchmarking programs. Opinion of the authors of this paper is that benchmarking programs currently used are too hard on scheduler, putting it in atypical working environment and testing only some of it's property. Therefore, more systematic approach is necessary. One such attempt will be presented in this paper. Identification activity of important scheduler properties can start with most discussed properties: scheduler efficiency, behavior under heavy loads and scheduling interactive tasks.

Testing for efficiency

Efficiency, in scheduler context, represents the ability of a system to perform more operations, to complete more tasks in less time using less system resources. For example, if a collection of jobs under some scheduler completes in less time than with second scheduler, then the first one is more efficient. To be efficient, scheduler should have small overhead and better scheduling decisions. Basic benchmarking of scheduling efficiency could be performed with collection of predefined tasks. Tasks may be independent, they can have dependencies (e.g., preceding-successor ordering), or they can use the same resources and synchronize themselves [4]. The benchmark could end when all tasks finish their jobs, or when predefined time elapses, in which case task's progress can be used for evaluation.

An example of extreme test for scheduler efficiency can be done with a set of tasks that exchange a token in a circular pattern [5]. Implementation of this benchmark could use pipes, messages or some synchronization mechanisms. Most of the time system will run in scheduler, because passing a token is a very short activity which is followed by activating next task that has received the token. This kind of tests will measure scheduler ability to switch between tasks.

Testing efficiency requires ability to start and simultaneously run a set of various tasks. E.g, tasks can communicate or not, continuously run or have sleep periods, have high priority, etc.

Testing for interactivity

Interactivity testing considers system's response times to user commands, input/output events, interactive applications, multimedia applications and similar. All those activities behave in a similar pattern: most of the time they do not require system attention (processing), but when they do, they should be processed as soon as possible. E.g., user may be annoyed by delayed reaction on his activities, like typing commands in shell, or using graphic interface. Multimedia applications may not have required quality if they experience long delays (sound might have pauses, video may skip frames). For desktop computers, which are mostly used by single user (at a time), scheduler should put precedence to user requests, and boost priority of foreground application when needed. For example, Microsoft Windows operating systems have a fairly simple scheduler [6], oriented toward single user systems[1] that satisfies requirements of foreground applications. Background applications get unused time, or at least bits of processor's time to avoid task starvations [7].

Testing for interactivity could be done with user who uses interactive applications in different working environments (e.g., different system loads). One problem with such approach is in quantification: how good or bad is the scheduler, how to compare schedulers, how scheduling changes with load? Another problem is in test repetition: how to repeat certain conditions that lead to particular system behavior.

Another way of testing interactivity is to eliminate the user from the test – construct test framework that will quantify scheduler behavior with synthetic interactive applications. Those tests might be more objective, but may also be too synthetic, i.e. they might miss to test some important scheduler behavior that occurs in real usage scenarios.

Synthetic tests can be used to evaluate several scheduler behaviors. The first one is regarding delays from events occurrence and start of their processing (in associated tasks). The second is in event processing durations (which can be interrupted and further delayed). Number of events processed too late may be indicator that scheduler is not performing as expected or system is not in normal state (e.g., heavy loaded).

Tasks used for interactivity testing could be modeled as periodic tasks, i.e. the ones that act on periodic events. Period length and event processing time could be test parameters. Many events in real systems do not arrive periodically but are

1 "Single-user system" in this context is used to describe system usage scenarios, not capabilities. System could have "multi-user" capabilities, but if in tested scenario is used only by single user, scheduler behavior will be evaluated as for single-user system.

aperiodic or sporadic. Still, using a periodic task or several with different periods such events can be simulated. In the end, system's reaction time is important for tests and that property should be investigated, not event occurrence patterns. Environment modeling can be extended with background tasks simulating different system loads.

Testing under different loads

Testing scheduler behavior under different system loads involves changing load and repeating tests. Since tests must be repeatable, even background load should be carefully modeled. Tasks for background load (*worker* tasks) could continuously consume all computing power when scheduled or use only part of given computing time. In a simple implementation this could be achieved with adding sleep interval that follows working interval. With duration of work and sleep intervals various background loads can be modeled. Other types of background tasks may include a set of tasks that intensively use scheduler, such as token passing tasks or even interactive tasks.

Background tasks priority can be lower, equal or even higher than priority of testing tasks.

Testing for scalability

Since multiprocessor systems have gone mainstream (as real multiprocessor or as multi-core) and are present in almost any new computer system, it is important to look into scheduler behavior in this environment. Testing schedulers for scalability requires no special changes in the test framework, but just adequate test parameters and appropriate result interpretation.

Scheduler must be aware of hardware characteristics and try to exploit them for achieving maximal performance. As an example, it is a common practice to associate tasks with particular processor, trying to exploit data already present in processors caches across successive task run time (interrupted by kernel calls and other tasks). However, if this is implemented by multiple run-queues, one for each processor, periodic balancing could be needed, preventing starvation and obtaining fair share of computing time for all tasks in system.

Testing for fairness

On a single-user systems, scheduler must satisfy requirements of a single user. Application currently used by the user (foreground application) is usually most important for user at that time, and should receive maximum system's attention. Scheduler should be aware of foreground application and raise its priority when it comes to foreground.

On multi-user systems scheduler must be fair. Fairness can be defined per user or per task. In *per user* concept, fair scheduler will balance different user's tasks execution times so that they distribute available computing time equally. In *per task* concept, all tasks should get fair (equal) amount of computing time. However, neither all users nor all tasks are equal. Some may be more important than other (e.g., “root” compared to a normal user). Differentiation is achieved trough different task (or/and user) priorities. For example, on UNIX like systems priority for non-real-time tasks is called *nice level*. The lower *nice* value is, a higher priority the task has. Values for nice are in range from -20 to 19. If the first task has nice of *n* and the second *n+1*, than the first task should receive more computing power than the second, e.g. 10% more [8]. Scheduler must take into account tasks priorities and accordingly distribute available computing power.

To test for fairness, test application should create task set, run it for some time, and compare tasks advancements. If tasks have same priorities, advancements should be similar, otherwise, task priorities should be seen in tasks progress.

II. TEST FRAMEWORK ARCHITECTURE

Scheduler testing should be performed with specific tasks. In a test framework it must be possible to define various tasks types. One way to achieve it, is to define generic properties which in combination define particular task type.

In a single test a collection of tasks can be used, some with same and some with different properties. Tasks with same properties can be grouped together and create a *task group*. Proposed configuration file skeleton is shown in Fig. 1.

```
global {
   test_name = “example”;
   duration = 60 s;
   run = interactive[1], background[3];
}
tasks {
   interactive {
      #definition of task properties
   }
   background {
      #definition of task properties
   }
} # comment from # to end of line
```

Fig. 1. Test configuration skeleton

Besides test name and duration, *global* section must define which tasks to create and run. For that purpose, group names and number of tasks to create are used, as shown in Fig. 1 with the *run* keyword. Task group properties are defined within *tasks {}* section, separately for each task group.

Defining worker tasks

Worker tasks come in two categories: in one are those that use all computing time, and in the other one are those that don't. Modeling the first group is done with a simple program loop, that will consume given computing time (processor's time). However, in order to measure task progress a number of finished loops can be counted. If a processing within the loop is too simple, a number of completed loops will be very large. It is preferred that the processing within single loop take measurable amount of time, e.g., one millisecond. In that way loop count will directly measure computing time a task had spent. Creating such job could be done by an inner loop that performs some simple operations a number of times. Since computing power differs across different processors, number of inner loop iterations (e.g., *loops_per_ms*) can be calculated in advance, before test is started. This could be performed in several ways.

The first approach could be to guess a number of iterations based on some system property, e.g., processor's speed. Execution of the calculated number of iterations should then be timed. Based on execution time, iterations could be corrected (even several times). However, some measurements must be

invalidated if timings are significantly longer – they are probably caused by task switch. Time measuring should be done with precise system calls, such as *clock_gettime()* [9].

In the second approach the same loop can be run for extended period of time. Then, using consumed *user time* retrieved from the system (e.g., with *getrusage()*) number of iterations that will yield a required time period for one inner loop can be calculated.

Both approach were tested and conclusion is that the second approach has a small advantage as it's results are little more consistent across several executions. But absolute accuracy is actually not needed. For comparing task progression its only necessary that jobs within iterations are the same, so that completed iterations can be used for comparison. Number of iterations only approximates task computation time, and its accuracy is less important for worker tasks. In test framework, although both methods were implemented, second approach is preferred. Before executing a test, a number of iterations for inner loop that will consume one millisecond time is defined in an environment variable and is used in the tests.

Defining worker tasks that don't work all time, require defining *working time* and *sleeping time* within single iteration [10].

Both worker task types can be in simplified form described with a pseudo code in Fig. 2.

```
worker_task {
   for iter = 0 to inf do {
      work(working_time);
      sleep(sleeping_time);
   }
}
```

Fig. 2. Worker tasks

Sleeping can be achieved in a number of different ways. System call *clock_nanosleep()* is chosen because of its high resolution. If a code like the one in Fig. 2 is used in test with more than one task from same task group then all the tasks will try to start execution at the same time, and then sleep at the same time, at least at the start of the test. Later they will probably desynchronize. However, it would be a good idea to desynchronize them intentionally at the start, with a randomization of their first working time duration.

Defining interactive tasks

As described before, interactive tasks can be modeled by periodic tasks. Periodic activity may be modeled by periodic signals sent to tasks, or tasks can suspend themselves until time of the next event (which can be precalculated). Both methods are used in real interactive applications, hence both should be used (separately) and tested.

Fig. 3 shows proposed implementation of interactive tasks using signal and sleep method.

Implementing periodic signals can be done in several different ways. A one solution is to use system call *setitimer()* which periodically sends signal to the calling task. With system call *pause()* task stops itself until signal is received. In signal handler function, times of start and end for event processing should be saved for later analysis (or analysis could be started in handler function).

```
interactive_task{
   if (interactive_signals_defined()) {
      set_signal_handler(signal_handler_function);
      create_periodic_timer(period);
   }
   if (interactive_sleep_defined())
      start_time = get_time();
   for iter = 0 to inf do {
      if (interactive_signals_defined())
         pause();
      if (interactive_sleep_defined()) {
         signal_handler_function();
         sleep_until(start_time + iter * period);
      }
   }
}
signal_handler_function() {
   save_start_time;
   simulate_event_processing;
   save_end_time;
}
```

Fig. 3. Interactive tasks with signal and sleep method

Without signals, interactive task should be delayed until next event occurrence. Clock used in *get_time()* and *sleep_until()* should be real time clock (or monotonic clock). Parameter to *sleep_until()* is absolute time, not relative. Relative waiting could result in prolonged periods due to eventual task switching. System calls that give required functionality and don't use signals are *clock_gettime()* and *clock_nanosleep()*.

For both implementations, “interactivity ratio” can be defined by defining period and event processing times. Real interactive tasks have period that is usually much longer than processing time.

Defining collaborative tasks

Collaborative (communicating) tasks represents tasks that communicate, exchange data or synchronize themselves, like token-passing communication. Collaborative tasks use scheduler more extensively than other task types, because scheduler is invoked much more frequently. Besides token-passing collaboration, another collaborative tasks widely used in multitasking applications are the ones that use shared resource protected by critical section mechanisms. Similar widely used task collaboration is with “barrier type” synchronization. In this synchronization all task are blocked at certain point in program until all other tasks reach the same point. Only then all tasks are released and may continue until next synchronization point.

All mentioned tasks have a start part, and an end part, as shown in Fig. 4.

```
cooperative_task {
   for iter = 0 to inf do {
      if (token_pass_defined()) {
         token_pass_wait();
         token_pass_process();
         token_pass_send();
      }
      if (critical_section_defined()) {
         critical_section_enter();
         critical_section_process();
         critical_section_leave();
      }
      if (barrier_defined()) {
         barrier_wait();
      }
   }
}
```

Fig. 4. Collaborative tasks

Defining additional task properties

Each task, even within the same task group, can have different priority. The simplest priority assignment is to set same priority to all tasks. A bit more complex assignment is to give priorities in a sequential order, e.g., first task gets priority *p*, the second *p*+1, the third *p*+2, etc. The priorities of the tasks and the assignment policy is a task group property (that is defined in the configuration file).

Single task can be represented by a thread in multi-threaded process, or as a separated process (with only one thread). Some systems may handle threads and processes in different way and the test framework, as specified here, support the evaluation of this difference (e.g., switching between threads from the same process might be faster than between threads from different processes).

Defining universal task

Universal task description should be able to describe all previous task types with their properties, as well as priority and representation (thread/process). Fig. 5 shows universal task code that includes all the behaviors from the previously discussed task types. Functions defined in the previous figures are not repeated since they are the same.

```
universal_task {
   initializations();
   for iter = 0 to inf do {
      iteration_specific();
      work(working_time);
      sleep(sleeping_time);
   }
}
initializations() {
   set_task_priority();
   if (interactive_signals_check()) {
      set_signal_handler(handler_function);
      create_periodic_timer(period);
   }
   if (interactive_sleep_check()) {
      start_time = get_time();
   }
   sync_start_for_all_tasks();
}
iteration_specific() {
   # per type processing, as in Fig. 3 and Fig. 4
}
```

Fig. 5. Universal task behavior

With the appropriate values of the task properties, various task types can be defined. E.g., for interactive tasks properties *working_time*, and *sleeping_time* should be set to zero.

An example of a task definition, including all the properties, of which are some commented out, is shown in Fig. 6.

```
tasks {
   example_task {
      working_time = 50 ms; # zero if not defined
      sleeping_time = 20 ms;
      priority = 5; # or e.g. "sequential[3, 16];"
      task_type = PROCESS; # (default) or THREAD
      #specific = INTERACTIVE_SIGNALS[250 ms,5 ms];
      #specific = INTERACTIVE_SLEEP[150 ms, 15 ms];
      #specific = TOKEN_PASS[MSG, 5 ms];
      #specific = CRITICAL_SECTION[5 ms];
      #specific = BARRIER;
   }
}
```

Fig. 6. Universal task definition

Proposed task definitions, both as in definition file and as task behavior defined with pseudo code, allow definition not of just previously mentioned task types, but also a task with mixed properties.

III. Selecting and defining tests

In the previous sections test conditions and scheduler properties that should be tested are defined. Also, test framework structure and configurations are presented. In this section few examples of concrete configurations are presented.

First test, as shown in Fig. 7, consists of a set of worker tasks with same priorities.

```
tasks {
   worker {
      working_time = 10 ms;
      priority = 5;
   }
}
global {
   test_name = "worker tasks";
   duration = 60 s;
   run = worker[10];
}
```

Fig. 7. Test configuration with worker type tasks

Using configuration from Fig. 7, test framework will create 10 tasks *(worker[10])* and run them for one minute (*duration = 60 s*). At the test's end, various statistic are reported, e.g., the number of iterations completed, used "user" and "system" times, number of context switches that were performed on task, task priority, task identification number like process and/or thread id, etc. (see Fig. 8). For a given test, comparing iteration numbers from all tasks should give us a measure of the scheduler fairness. Also, different schedulers could be used on same system, they can be compared for efficiency (comparing iterations).

The tests were performed on *Ubuntu* distribution with Linux kernel 2.6.31 (x86_64), with one to four processors (cores) set to constant (maximal) processor's frequency (and hyper-threading ability switched off). Test results for configuration from Fig. 7 (not given here in raw form) showed that scheduler is satisfactory fair in distributing available computing time. When only one processor was activated, tasks got almost the same computing time (numbers of iterations are very close). When more processors are activated, tasks iterations are mostly close to mean value. However, few tasks may have significantly lower or higher values for iteration, with a deviation of up to 25 percent (from several experiments). Since tasks are independent, with four processors number of iterations was four time larger – compared to test with single processor.

When processors' speed was set to "on demand" and hyper-threading ability was activated (default settings!), test showed major progress difference between tasks, indicating scheduler is not yet appropriately adapted for such hardware environment (in respect to fairness).

Changing priorities assignment to sequential will show how initial task priority contribute to given computing time. If the configuration from Fig. 7 is modified by changing line 4 to:

priority = sequential[1-10];

then the first created task will get priority of one and the last one will got the priority ten. The result from a single execution of the test is presented in Fig. 8 and Fig. 9. Repeating the tests shows similar results.

```
TN  Name    PID    NICE ITERS     UTIME(C)  UTIME     STIME    CW(v/i)
-------------------------------------------------------------------
0   sample  2352    10  5454      54.540000 55.260000 0.000000 1/1740
1   sample  2353    11  4487      44.870000 45.400000 0.000000 1/1335
2   sample  2354    12  3195      31.950000 32.280000 0.000000 1/1062
3   sample  2355    13  2489      24.890000 25.460000 0.000000 1/892
4   sample  2356    14  2064      20.640000 21.200000 0.000000 1/924
5   sample  2357    15  1554      15.540000 15.730000 0.000000 1/749
6   sample  2358    16  1395      13.950000 14.360000 0.000000 1/840
7   sample  2359    17  1169      11.690000 11.870000 0.000000 1/723
8   sample  2360    18  1099      10.990000 11.140000 0.000000 1/770
9   sample  2361    19  673       6.730000  6.920000  0.000000 1/641
'sample':  iter_sum=23579 (235.79 s), avg=2357, stdev=1575
```

Fig. 8. Test results for worker tasks with different priorities

Column *UTIME(C)* presents calculated run time based on iterations completed and assuming accurate inner loop timings. Columns *UTIME* and *STIME* present data for "user" and "system" time respectively, collected using *getrusage()* system call. Difference between *UTIME(C)* and *UTIME* comes from inaccuracy from both inner loop calibration and inaccuracy of the statistic collected by the system (kernel). The last two columns *CW(v/i)* contain numbers of context switches performed on task, both voluntary (when task blocks itself, like blocking on semaphore) and involuntary (task was switched by scheduler).

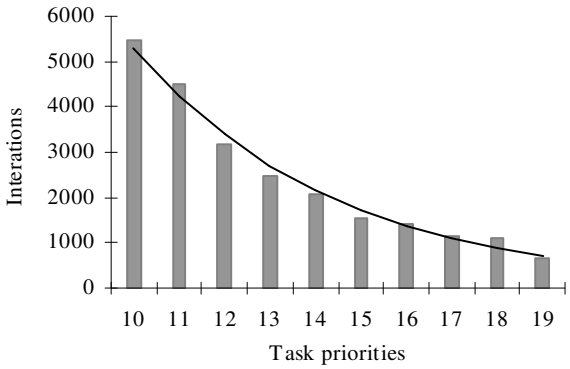

Fig. 9. Priority and progress relation

Bars in the Fig. 9 show the number of iterations achieved by all tasks. Since each task has different priority, each task progress was different. Comparing tasks with priority difference of one and their iterations, an average ratio of 1.25 was calculated in favor of higher priority task (lower "nice" number). Line above bars in Fig. 9 shows trend-line if ratio is to be exactly 1.25. The results show good conformity with trend-line. However, not all tests gave such results. Tests with shorter period on multiprocessor have larger deviations from trend-line, although globally the same ratio is followed.

Introducing sleep time into previous configuration, tests become less intensive on the scheduler, but the relative results are similar: fairness and respect for different priorities didn't change.

Tests with collaborative tasks can show several scheduler properties. With tasks that intensively collaborate, context switching ability can be measured. The example configuration for such test is shown in Fig. 10.

```
tasks {
   token_pass {
      working_time = 0 ms;
      priority = 5;
      specific = TOKEN_PASS[MSG, 0 ms];
   }
}
```

Fig. 10. Token pass task

Section *global* is not shown in the Fig. 10 because it is similar to the same section given in the Fig. 7. Working time is set to zero both for task iteration and token processing, forcing task to block immediately after passing token on (messages were used for passing token). Number of instructions task perform in single iteration before it's switched is almost negligible compared to task switching time and scheduler activity, which are in focus.

With critical section and barrier synchronizations, scheduler can also be tested in extreme conditions, putting all working times to zero. However, those tests are also suitable for evaluating possible gains from multiprocessor systems when a job can be executed by collaborative tasks. Also, with proper distribution of tasks across multiprocessor, scheduler can help in improving performance of such applications. For example, when collaboration can be defined by critical section synchronization, and at some point in application execution only two non-blocked tasks are present on the same processor (one as active and second as a ready task), it would be desirable that the scheduler moves the second task to another processor. It could be that first (active) task has just exit critical section and unblocked second one (that is scheduled on same processor). First task could spend a majority of its current time slice in non-critical section, further delaying second task from executing its critical section, and releasing other task that could run on different processor. This switch should be done regardless of other tasks in system (tasks from other applications). However, such scheduler behavior shouldn't be coded per situation basis, otherwise, a lot of heuristic can also be coded in scheduler, who could became more complex, slower and possibly not work for all situations. But still, since multiprocessors are reality, some simple heuristic regarding scheduling a multi-task application should be applied if possible.

Interactivity behavior can be tested with previously defined interactive task. Two main parameters define interactive task: period in which event recur, and event processing time. In Fig. 11 definitions of two tasks were present: first one activated with signal and second one which is waked on timer expiration.

```
tasks {
   interactive_signal {
      specific = INTERACTIVE_SIGNALS[150 ms, 5 ms];
   }
   interactive_sleep {
      specific = INTERACTIVE_SLEEP[100 ms, 15 ms];
   }
}
```

Fig. 11. Token pass task

Tests on low loaded system, with configuration as in Fig. 11, showed very good scheduler interactivity behavior. Also, shortening period (even down to 1 millisecond on tested system) didn't increase reaction delay. Loading system with moderate

"background", reactions becomes little slower, but as long as their period is long enough (at least 10 milliseconds) and background tasks have lower priority, reactions are tolerable.

With heavier system loads overall delay in event processing grows. It depends on application if this longer delays (e.g., 10 to 20 milliseconds) are still acceptable. In example, if user uses multimedia application to play music while working on something else, regardless of that work, music should be played without degradation in quality. Since music player uses only small percent on computing power, scheduler should give him processor's time as soon as requests is made.

IV. Conclusion

Evaluating most active part of an operating system – its task scheduler – is a challenging problem. First, the required scheduler behavior must be defined for all (most) usage scenarios. Second, methods for testing scheduler in those scenarios, methods that are also realizable within given system and its interfaces, must be designed and implemented. Limitations from both: used interfaces and simplifications, must be considered when analyzing test results.

In this paper, an attempt to define required scheduler behavior is presented and elaborated. Based on the presented ideas the test framework is designed and using test framework different basic tests are defined and performed, with brief comments of obtained results.

On the implementation side several problems were encountered. Interface used in the test framework is chosen to be compatible with most Linux based systems. However, precision of those interfaces, as well as behavior, slightly changes across different kernel versions.

The first implementation problem was how to create simulated work with precisely timed execution (determine number of loop iterations that give required duration). Timing is only required for creating tasks with given work/sleep ratio, for defining work time for collaborative and interactive tasks. Timing simple loops showed to be unreliable method since it can be interrupted by the scheduler for a short or even longer time (e.g., task switching). Using longer loop and collecting used "user" time from system relies on imprecise system interface. However, the later approach turned out to be more consistent in repeated tests and is thus preferred method. Based on test experiences, its accuracy its approximately within 10%, which is satisfactory for tests.

The next problem was how to implement "sleep" period. On some systems the interface used (*clock_nanosleep*) showed to be quite accurate. On the others, it wasn't so accurate, probably due to a different time slice and a minimal sleep period. On those systems, if test results are to be meaningful and sleeps are required, longer periods should be used.

Time measurement interfaces suffer from the same problems as implementation of sleep – somewhere are accurate down to 1 millisecond and below, somewhere lower bound is higher.

The test framework was built for scheduler testing only and could be inadequate for other system tests, like performance evaluation. It should be noted that simple program changes (e.g., type and scope of a variable) can noticeably change loop number a task can execute within same time period. Utilized simulated work in (inner) loop includes simple logic operations on integer numbers, and may not relate for real tasks that perhaps use float point arithmetic or are memory intensive. However, approximate comparison between different systems, from performance perspective, can be made (using the same number of inner loop iterations – bypassing calibration).

Based on tests performed during and after building test framework, the results roughly follow expectations. On multiprocessor, fair division of computing time its harder to achieve. Interactive application are treated better than expected (based on past experience), but that could be in part because of the tested system (newer kernel and hardware).

Future work will include extensive testing with the constructed framework on more diverse hardware/software systems, creation of test configuration that more accurately model real working scenarios and test particular scheduler property or more of them simultaneously.

Someone could ask why putting such effort on building and testing schedulers since programmer can significantly influence application scheduling by setting appropriate priority and scheduling policy. E.g., for an interactive application, a real-time scheduling could be chosen, removing problems possibly caused by other tasks. That could be true, but first, running such application would require higher user privileges, and second, more important, adding scheduling issues to process of software development would further complicate already complex process. Complexity is primary reason for software errors and poor quality in today's applications, and should not be further boosted. That applies to schedulers as well.

References

[1] J. Aas, "Understanding the Linux 2.6.8.1 CPU Scheduler," *Silicon Graphics, http://joshaas.net/linux/*, February 2005.

[2] I. Molnar, "Completely Fair Scheduler," *http://kerneltrap.org/node/8059*, April 2007.

[3] J. Andrews et al, "Additional CFS Benchmarks," *http://kerneltrap.org/Linux/Additional_CFS_Benchmarks*, September 2007.

[4] L. Budin, "Operating systems," *Element Zagreb*, in press.

[5] "Pipe-test", http://kerneltrap.org/node/14745.

[6] Win32 thread scheduling, *http://msdn.microsoft.com/en-us/library/ms685096(VS.85).aspx*, October 2009.

[7] B. Hunjadi, "Process scheduling in Win32 environment," *undergraduate thesis (in Croatian), FEE&C – University of Zagreb*, June 2009.

[8] J. Andrews et al, "Linux: History Of Nice Levels," *http://kerneltrap.org/node/11778*, July 2007.

[9] The Single UNIX Specification (man pages), *http://www.opengroup.org/onlinepubs/009695399/.*

[10] Con Kolivas, "Ten percent test," *http://lkml.org/lkml/2007/4/5/384*, April 2007.