Latencies in Linux and FreeBSD kernels with different schedulers – O(1), CFS, 4BSD, ULE

Jaroslav ABAFFY
Faculty of Informatics and Information Technologies
Slovak University of Technology
Bratislava, 842 16, Slovakia

and

Tibor KRAJČOVIČ
Faculty of Informatics and Information Technologies,
Slovak University of Technology
Bratislava, 842 16, Slovakia

ABSTRACT

This paper is a study of scheduler latencies in different versions of Linux 2.6 kernel with emphasis set on its usage in real-time systems. It tries to find the optimal kernel configuration leading to minimal latencies using some soft real-time tuning options. Benchmark tests under heavy load show differences between kernels and also between different scheduling policies. We compare Linux kernel 2.6.22 with a long-acting O(1) scheduler with the to date latest Linux kernel 2.6.28 that already uses a completely new CFS scheduler. Not only a scheduler, but also other kernel options can lead to latency reducing. We compare a kernel compiled for server employment and then tuned to act as a soft real time operating system. For better comparison we perform selected benchmarks also on FreeBSD 7.1 kernel compiled with the older 4BSD scheduler and with the newer ULE scheduler. ULE scheduler was improved in the version 7 of FreeBSD, so we compare it also with ULE in FreeBSD 6.3. The emphasis of this paper is set on finding a scheduler with minimal latencies on tested hardware.

Keywords: Latency, Linux, FreeBSD, Scheduling, Benchmark, Kernel.

1. INTRODUCTION

Linux and FreeBSD were developed as general purpose operating systems without any consideration for real-time applications and are widely used as server operating systems. In recent years they have become attractive also as desktop operating systems, and nowadays they find their way to the real time community due to their low cost and open standards. There is a big dilemma not only in programming operating systems called throughput vs. latency. There is a large throughput expected in servers, but in embedded systems the main goal is low latency.

Methodology used for finding optimal kernel configuration is based on setting relevant kernel options, compiling the kernel, and running benchmarks on it. With this method we are able to acquire the optimal kernel with CFS scheduler and also with O(1) scheduler. These kernels are compared with one another and also compared with versions compiled in default configuration. For FreeBSD we use default configuration, but once with 4BSD and then with ULE scheduler.

The benchmark primary used for comparison of these kernels was Interbench. During the tests we found out that this benchmark can under several conditions cause inaccurate results. That provoked us into developing other benchmark called PI-ping. With this tool we are able to compare latencies under heavy load and are also able to explain previous misleading results. Our results were then approved by another benchmark – Hackbench.

PI-ping is based on two different types of process that occur in operating systems. In the first group there are processes that demand a lot of CPU. Other processes are interactive – they request good latencies. In this benchmark we use for the first group a computation of the number Л, and as interactive process we use network ping to localhost every 100 ms. This time was already a long time ago empirically determined as the maximum latency when the user considers the system as interactive. Therefore it is widely used in different operating systems as a default amount of time assigned by the scheduler to the process between two task switches. Computation of the number Л is used to prevent any optimization by the compiler because the ciphers are not predictable. With this tool we are able to compare how well the kernel and the scheduler is desirable for CPU consuming processes,

and we can also see how many interactive process can under heavy load meet their deadlines.

2. TESTED KERNELS

Linux 2.6.22 with O(1) scheduler
This kernel is the latest Linux kernel that uses O(1) scheduler. The name of the scheduler is based on the popular big O notation that is used to determine the complexity of algorithms. It doesn't mean that this scheduler is the fastest, but it means that it can schedule processes within a constant amount of time independent on the number of tasks running in the system. Therefore, this scheduler is suitable for real-time application because it guarantees the highest scheduling time.

Linux 2.6.28 with CFS scheduler
Since the version 2.6.23 Linux kernel comes with Completely Fair Scheduler (CFS) that is the first implementation of a fair queuing process scheduler in widely used general-purpose operating systems. Schedulers in other operating systems (and also O(1) scheduler) are based on run queues, but this scheduler arranges processes in a red-black tree. The complexity of this scheduler is O(log n).

The main advantage of a red-black tree is that the longest path in this tree is at most twice as long as the shortest path. This scheduler was written to accept different requirements in desktops and in servers.

FreeBSD 7.1 with 4BSD scheduler
4BSD is the default scheduler in all FreeBSD versions prior to 7.1, although there is a new scheduler called ULE since FreeBSD 5.0. It is the traditional Unix scheduler inherited from 4.3BSD, but in FreeBSD there were added scheduling classes. It is also based on run queues as the Linux O(1) scheduler.

FreeBSD 7.1 with ULE scheduler
The name of this latest scheduler in FreeBSD comes from the filename where is it located in source code of the kernel /sys/kern/sched_ule.c. In comparison to O(1) and 4BSD there are not two run queues, but in this case three: idle, current and next. Processes are scheduled from the queue current, and after expiration of their time slice they are moved to the queue next. Rescheduling is made by switching these two queues. In the queue idle there are idle processes.

The main advantage of this scheduler is that it can have run queues per processor, what enables better performance results on multiprocessors. In this paper we perform selected benchmarks also on older ULE scheduler from FreeBSD 6.3 to see if there were made significant improvements as presented in [7].

3. INTERBENCH BENCHMARK

For testing different kernels we used a program named Interbench that generates system load and measures latencies under different conditions. Tested interactive tasks are:

- Audio - simulated as a thread that tries to run at 50ms intervals that then requires 5% CPU (20 times in a second)
- Video - simulated as a thread that uses 40% CPU and tries to receive CPU 60 times per second.
- X - simulated as a thread that uses a variable amount of CPU ranging from 0 to 100%. This simulates an idle GUI where a window is grabbed and then dragged across the screen.
- Server - simulated as a thread that uses 90% CPU and tries to run at 20ms intervals (20 times in a second). This simulates an overloaded server.

These tasks were tested under different system loads:
- None - otherwise idle system.
- Video – the video simulation thread is also used as a background load.
- X - the X simulation thread is used as a load.
- Burn – 4 threads fully CPU bound.
- Write - a streaming write to disk repeatedly of a file the size of physical ram.
- Read - repeatedly reading a file from disk of the size of physical ram.
- Compile- simulating a heavy 'make -j4' compilation by running Burn, Write and Read concurrently.
- Memload - simulating heavy memory and swap pressure by repeatedly accessing 110% of available ram and moving it around and freeing it.
- Server - the server simulation thread is used as a load.

Each test was performed for 30 seconds and used 1 055 301 CPU cycles per second, so it can be considered as a sufficient time to obtain relevant data. The whole test took ca. 20 minutes.

Soft real-time kernel options
For finding a kernel configuration that leads to minimal interrupt latency we used as a reference kernel the kernel in default configuration, and then we were adding some relevant options in kernel configuration. This tuned kernel was compared to the default configuration to see if it has improved the real-time performance. Best results were achieved using these kernel options:

- Dynamic ticks
- High Resolution Timer Support
- Timer frequency 1000 HZ
- HPET Timer Support
- Preemptible Kernel
- Preempt The Big Kernel Lock

Kernel compiled with these options is in this paper called soft real-time – SRT. Kernel without these options is called Server because it desires better throughput instead of low latencies.

Interbench results

In following graph (Fig .1) there are shown the average latencies, their standard deviations, and maximal latencies the simulated video thread under different system loads when using the default configuration of Linux kernel 2.6.22. We tested all mentioned tasks, but in the video thread benchmark there are the differences most visible.

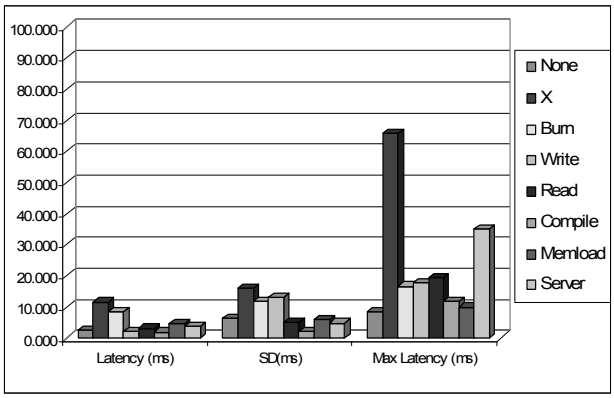

Fig. 1. Latencies in kernel 2.6.22 – Server (lower is better)

After compiling the kernel using soft real-time kernel options we were able to achieve improvements in these latencies (Fig. 2). Only under the load X which represents a user activity in graphical interface, there is degradation especially for the maximal latency. This can be explained so that in the time when the benchmarked video thread required the CPU also the X thread demanded a lot of CPU. Video thread requires 40% of CPU every one second, but X demands variable amount in random times.

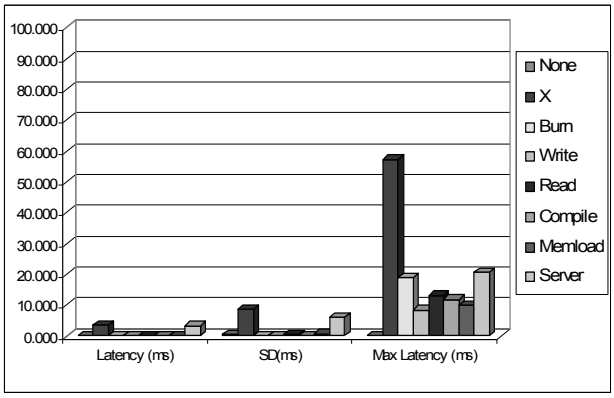

Fig. 2. Latencies in kernel 2.6.22 – SRT (lower is better)

In the next graph (Fig. 3.) there are the results for Linux 2.6.28. Using different kernel options did not lead in this case to such observable impact on scheduling latencies, the reason is explained by the author of this scheduler Ingo Molnar: "CFS uses nanosecond granularity accounting and does not rely on any jiffies or other HZ detail. Thus the CFS scheduler has no notion of 'timeslices' and has no heuristics whatsoever."[5]

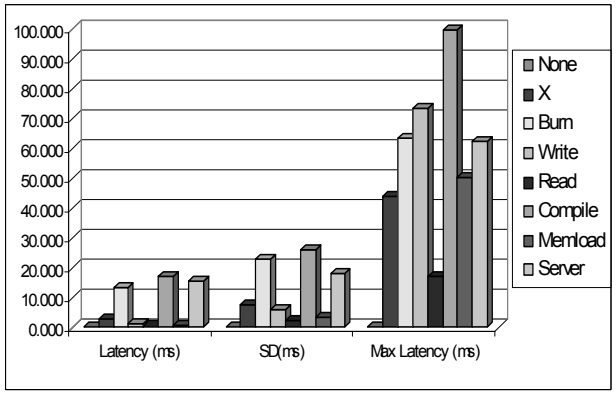

Fig. 3. Latencies in kernel 2.6.28 (lower is better)

GFS scheduler

Benchmarking these kernels using Interbench has shown big differences between Linux 2.6.22 and 2.6.28. The main question that appeared is why was CFS scheduler included into the main kernel when it has such performance overhead compared to O(1) under almost all conditions. CFS defeats only when no other load is in the system, or when the load simulating graphical user interface is presented.

CFS should have perform with low latencies especially under the loads when some processes are interactive, and the other are CPU demanding. In our case these CPU demanding processes are the loads Burn, Compile, and Server where CFS theoretically has to have better results, but practically these were the loads where it has its biggest problems.

Problem with Interbench is that it runs benchmark and load as threads within one process. CFS scheduler tries to distribute the resources fair between processes, and parent processes share their assigned time quantum with their children. In the case when Interbench uses CPU demanding thread as a load and benchmarks interactive thread like video, CFS scheduler divides the quantum between these threads.

In the next table (Tab. 1.) we run Interbench thread Video with no other load from one shell and the load Burn representing 4 processes demanding 100% CPU from another shell. This load was selected because it depends mostly on CPU; other loads use also a lot of memory and IO operations. Different kernel versions can have different IO schedulers and memory management, and we target now on the scheduler.

Tab. 1. Latencies with GFS

	Latency (ms)	SD (ms)	Max Latency (ms)	% Desired CPU	% Deadlines Met
O(1)	0.021	0.436	16.70	100.00	99.90
CFS	20.700	25.300	59.5	69.70	19.30
CFS – 2 shells	16.200	20.500	36.20	83.80	32.90
CFS + GFS	0.007	0.008	0.018	100.00	100.00

An extension called Group Fair Scheduler (GFS) was added to the CFS scheduler in Linux kernel 2.6.24. It enables fair CPU sharing between groups of processes, in this case based on user ID. If user A runs 100 processes and user B 5 processes, both of the users become 50% of CPU independent on the number of processes. So we run the benchmark under two different users and achieved so already better results than with O(1) scheduler.

Interbench problems

When using Interbench, we found out several problems. It benchmarks only one thread and says nothing about the threads that are used as load. When we run Interbench with 100 load processes in Linux 2.6.22, the system was very slow, and the test took 60.261 seconds compared to 32.537 seconds in 2.6.28. Also interaction to user inputs was very slow, but the results said something else. Interbench is available only for Linux, and we wanted to compare Linux and FreeBSD kernels. This motivated us to create an own benchmark.

4 PI-PING BENCHMARK

Pi-ping uses two types of processes for benchmarking which appear in operating systems. One of them are interactive tasks demanding low latencies, the other group are processes with high CPU utilization. Modern operating systems have to perform well under both conditions.

In the following graph (Fig.4.) there is shown how many deadlines were met by an interactive process ping. The calculation is simple, we run ping every 100 ms and measure the time of the whole benchmark rounded up to 100 ms. After dividing this time by 100 we obtain the number of expected successful pings. The ratio of measured pings compared to the number of expected says how many deadlines were met.

Most of the benchmarks are designed either for latency measurements or for performance comparison, but we wanted to compare both of them. As

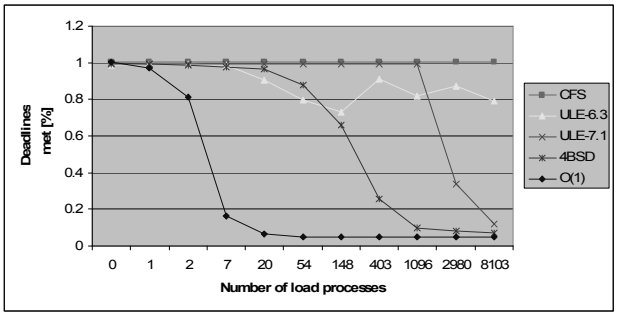

Fig. 4. Percentage of met deadlines in dependence of the number of load processes (higher is better)

Using this benchmark we obtained completely different results than in using Interbench. CFS succeeded already under the load of more than 8000 processes, and always 100% of the interactive processes met their deadlines.

O(1), 4BSD and ULE-7.1 have similar characteristics, but be aware of the logarithmic axis used in this graph. The latest ULE scheduler in FreeBSD 7.1 can also be considered as a low latency scheduler since it also covers 100% deadlines when the number of load processes is in acceptable limits. Already servers are often limited by administrators to maximal 1024 processes, and now we focus on latencies in small systems.

Surprising was the decrease of O(1) scheduler, by only 2 load processes running in background the ping responsiveness was only around 80%. In Interbench benchmark we achieved with the same kernel 99.9% coverage of deadlines by 4 load processes using Video thread as test which demanded 40% of CPU. How is it possible that CPU consuming test scores better than a small ping process?

PI-ping shows also problems with ULE scheduler in FreeBSD 6.3. Results in the graph (Fig.4.) are the average values measured in 3 benchmarks. Other schedulers have balanced characteristic, and the results were almost same in each experiment. In following graph there are the results of three measurements of ULE-6.3 scheduler.

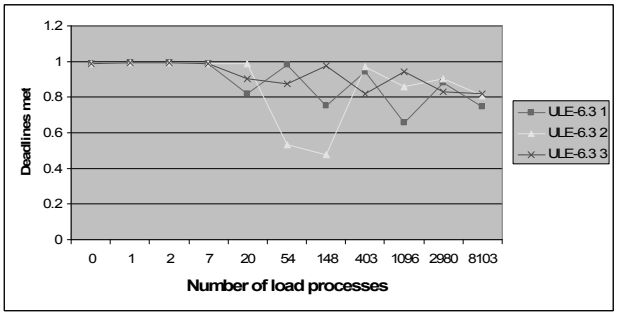

Fig. 5. Results of ULE-6.3 in 3 measurements

As you can see, by raising the number of processes the results of ULE scheduler become in FreeBSD 6.3 instable.

We also wanted to perform benchmark not only for interactive tasks, but also for CPU demanding processes like computation of the number Л. In the following graph CFS is used as reference scheduler and the speed of other schedulers is calculated as the time of CFS divided by the time of the other scheduler.

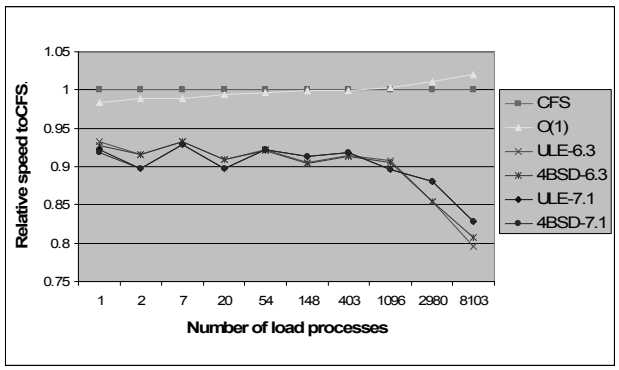

Fig. 6. Relative speed of Л computation compared to CFS (higher is better)

In this test FreeBSD is approximately 10% slower than Linux. It can be caused by optimization because Linux was compiled for i686 and FreeBSD for i386, by the different implementation of task switching routine, by the concept of forking processes, or by any other kernel options than scheduler. Important is that for the computation of Л the scheduler doesn't have high impact. These processes do not use any shared memory, pipes, mutexes, lot of IO operations et cetera and always use the whole time slice assigned by the scheduler which is 100 ms by default in the both operating systems. If we run for example 10 processes, each running for 10 seconds, the computation should take 100 seconds in ideal case. But in the real case it is higher because of the overhead caused by operating system, rescheduling, and other running processes in the system.

When comparing FreeBSD with ULE scheduler and with 4BSD, the results are almost the same. Small differences are only between different versions (7.1 versus 6.3), but the curves look similar. ULE and 4BSD are based on run queues with the complexity O(1), so for these non-interactive tasks they perform equal. But in case of Linux, we can see the confrontation of the CFS scheduler with O(log n) complexity with the older O(1) scheduler. For smaller numbers of processes CFS performs better, but if there are a lot of processes, O(1) takes advantage of its better complexity. In this case the intersection of O(1) and O(log n) was experimentally set at the point, where 500 processes are in the system.

5 HACKBENCH BENCHMARK

In previous benchmark we have inspected instable results of ULE scheduler in FreeBSD-6.3 and demonstrated that CFS performs better for interactive processes and has also better results for non-interactive processes when there is not too much of them.

To approve the results of PI-ping benchmark we used another test called Hackbench. This benchmark launches a selected number of processes that either listen on a socket or on a pipe and complimentary the same number of processes that send 100 messages to the listening processes.

The results of the benchmark are in the graph (Fig.7.), but now we use not average values, but from only one benchmark, to depict the differences in stability of ULE-6.3 and ULE-7.1. In this benchmark ULE-6.3 performs better for smaller number of processes, but then becomes the instability predicted by PI-ping visible. CFS, O(1) and ULE-7.1 have linear characteristic.

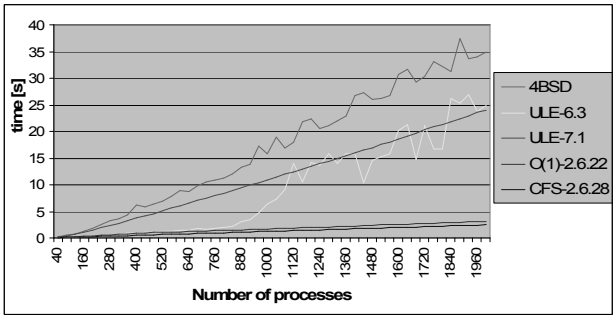

Fig. 7. Results of Hackbench benchmark (lower is better)

The curves of CFS and O(1) are very near to each other. To make it more visible, we show the results as the relative latency per process to CFS (Fig.8.). We can see, that O(1) nearest very slowly to CFS. In case of Pi-ping, the Л processes were CPU demanding, Hackbench processes are pure interactive. That is the reason why in PI-ping by already 500 processes O(1) performed better then CFS. CFS is designed to favoritism small and fast processes.

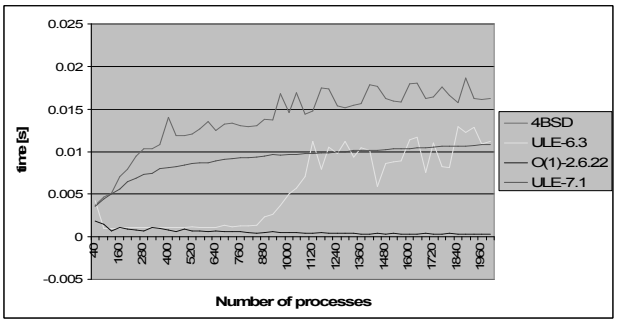

Fig. 8. Relative latency per process compared with CFS (lower is better)

6. TESTING HARDWARE

For testing and benchmarking was used a common laptop with 1 GB RAM and Celeron M processor at 1.6 GHz. Important is that the used processor is single-core. Using multi-core processor with enabled symmetric multiprocessing would affect the results in significant way.

7. CONCLUSION

The results and graphs show that the new Linux scheduler CFS really competes in both employments – in a server field demanding high throughput and also in embedded systems demanding low latencies. Other schedulers compared in this paper are based on run queues; this one organizes processes in a red-black tree. Computational complexity of other schedulers is O(1), CFS has the complexity O(log n). We have shown that CFS performs better than O(1) for computational tasks when the number of processes is smaller than approximately 500. This was by us experimentally defined as the intersection of O(1) and O(log n) functions in this case. But for interactive tasks it performs better in all tested situations.

The goal of this work was to show that the complexity O(log n) of the CFS scheduler is not a handicap for real applications, and we can recommended also for embedded systems demanding real-time performance.

We have also shown the improvement of ULE scheduler in the latest version of FreeBSD. ULE in FreeBSD 7.1 performs better than long acting 4BSD and does not suffer the problems inspected by using this scheduler in FreeBSD 6.3.

8. ACKNOWLEDGEMENT

This work was supported by the Grant No.1/0649/09 of the Slovak VEGA Grant Agency.

9. REFERENCES

[1] Benchmark Programs. http://elinux.org/Benchmark_Programs [27.01.2009]

[2] Lukas Jelinek, **Jadro systemu Linux**, Computer press, 2008

[3] Sanjoy Baruah and Joel Goossens, "Scheduling Real-Time Tasks: Algorithms and Complexity", **Handbook of Scheduling: Algorithms, Models and Performance Analysis**, Chapman & Hall/CRC, 2004

[4] Clark Williams, Linux Scheduler Latency http://www.linuxdevices.com/articles/AT8906594941.html [27.01.2009]

[5] Ingo Molnar, This is the CFS Scheduler http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt [27.01.2009]

[6] Gilad Ben-Yossef, "Soft, hard and ruby hard real time with Linux http://www.scribd.com/doc/3469938/Soft-hard-and-ruby-hard-real-time-with-Linux [27.01.2009]

[7] Kris Kennaway, "Introducing FreeBSD 7.0" http://people.freebsd.org/~kris/scaling/7.0%20Preview.pdf [27.01.2009]

[8] Jaroslav Abaffy, "Interrupt Latency in Linux 2.6", **Informatics and Information Technologies Student Research Conference**, Vydavatelstvo STU, 2008, pp. 387-393