Paper

Linux Scheduler Improvement for Time Demanding Network Applications, Running on Communication Platform Systems

Marcin Hasse and Krzysztof Nowicki

Abstract—Communication platform systems as, e.g., advanced telecommunication computing architecture (ATCA) standard blades located in standardized chassis, provides high level communication services between system peripherals. Each ATCA blade brings dedicated functionality to the system but can as well exist as separated host responsible for servicing set of task. According to platform philosophy these parts of system can be quite independent against another solutions provided by competitors. Each system design can be different and can face with many computer systems design problems. One of the most difficult design problems to solve is system integration with a set of components running on different operating system levels. This paper presents Linux scheduler improvement possibility to make user space application classified as time demanding (required to be serviced by CPU in given amount of time) running in user space together with complicated kernel software structure in the system.

Keywords*— *communication platform systems, Linux, operating system, scheduler.

1. Introduction

Today's communication trends are consolidated to follow platform strategy. This strategy is to provide standard base solutions to be re-used over wide range of products. Advanced telecommunication computing architecture (ATCA) [1], [2] is a very good example which is successfully matching platform objectives. The communication architecture between subsystems, the major issue in platform implementation, seems to be prepared to manage restricted subsystems requirements. Standard base chassis with intelligent platform management interface (IPMI) and Ethernet communication makes a very friendly base for a big range of network products like switches and gateways as well as for computer base products like single board computers (SBC). Well organized communication between subsystems and advanced management opportunities makes ATCA platform very interesting solution for telecommunication market, especially because these systems are following restricted energy consumption and thermal norms.

From the other perspective, systems prepared to match ATCA standard have a big challenge to follow restricted norms and propose good enough performance for end users. The law formed by Herb Grosh in 1965 [3] indicating that computer performance increases as the square of the cost. Regarding to this law SBC with more RAM memory and with bigger HDD would have a better performance, but ATCA system performance can not be limited only to regular PC specific costs. They need to be considered as well energy and thermal system assumptions which makes the cost more significant. This is causing that ATCA systems are designed with limited system resources mostly according only to design demands.

Platform strategy gives opportunity to application designers to choice ATCA hardware base on system demands. For example Ethernet line card hardware (based on network processor or other multicore processor) would be a good choice for IPsec gateway application.

Only one disadvantage of customize hardware to match ATCA platform standards is that blades (as line cards) can not easily be extended to additional system resources as RAM or flash memory. Application designers are responsible for achieving software goals with available resources starts from operating system and ends on specific application (as IPsec IKE [4], [5] for IPsec gateway example).

Linux is a most popular platform choice for ATCA blades used in network core: as gateways, routers, etc. It would be as well most reasonable choice for IPsec gateway Ethernet line card example. There are several Linux operating systems available with embedded system support and with ATCA blades board support packages (BSP) as Montavista [6] or WindRiver [7]. Additional advantage of Linux OS is its open source nature and developers have access even to kernel sources. This is big opportunity to have more influence on system performance while developer can place program in the kernel level. Regarding Linux GPL [8] licence programs in kernel space suppose to be published as open source. This restriction creates a barrier for Linux commercial application providers – which are mostly offered as a user space programs. For example additional IPsec gateway functionalities as virtual router redundancy protocol (VRRP) [9] or simple network management protocol (SNMP) [10] can be taken from independent supplier as a user space application.

This paper indicates problems with limited system resources operated by Linux OS and common problems with user and kernel space applications working together in net-

work and real time environment. In Section 2 there will be Linux scheduler analyzed in order to present issue with time demand user space application working together with real time tasks in kernel level. Proposed scheduler improvement to make Linux more flexible if there are time demanding user space applications is described in Section 3. Measurement of improvement results plus comparison with standard scheduler, are described in Section 4.

2. Linux Scheduler against User Space Time Demanding Processes

In the Linux operating system there can be determined two kinds of threads [11]. First would be CPU bound, which spends a lot of time using central processing unit (CPU) and making computation. The second would be I/O bound most time waiting for a I/O operation to complete. Scheduler in Linux is designed to deal with both types of threads in the fair way, but there is no well known method to determine if thread should be classified as I/O bound or CPU bound. The reason why scheduler should tread I/O bound threads with bigger priority is slow nature of I/O. There is understandable requirement to service human input as fast as possible – most people simply do not like wait especially when they wanted to have something done by a computer. It takes a long time for service I/O so it is good if that kind of requests can be serviced as fast as possible.

Linux scheduler goals as efficiency and interactivity makes this mechanism more friendly for servers (most common usage of Linux these days) and for desktop (where Linux would like to be more important than today). Unfortunately, if something is more matching servers and desktops then it is probably less matching core network systems as, e.g., gateways.

In order to evaluate scheduler role in the system it would be efficient to determinate scheduler performance. Introduction this metric should allow checking if scheduler works properly for given set of margin conditions (different than for normal server or desktop usage conditions). In most cases performance determines the time required to finish the task. For process scheduler performance it would be time in which task (CPU or I/O) will be successfully serviced. In the other words performance P (for the process with priority X), would be a process wait time until it will be serviced by CPU T_w and CPU execution slice time T_s with assumption that task could not be finished in Q CPU slices:

$$P(P_X) = \sum_{n=1}^{Q} \Big[T_w(P_X) + T_s \Big].$$

Waiting queue T_w time is dependent on several additional systems conditions as number of tasks N waiting for CPU and their priorities time slice T_{sX}:

$$T_w(P_X) = \sum_{n=1}^{X} (N \cdot T_{sX}).$$

While there will be several the same priority tasks, e.g., S) for scheduler it will service them in request order:

$$P(P_X) = \sum_{n=1}^{Q} \left\{ \sum_{k=1}^{X} \left(N \cdot ST_{sX} \right) + T_s \right\}.$$

Priorities in Linux kernel 2.6 scheduler can be set between 0 and 139, where priorities between 0–99 determine kernel threads and 100–139 determine user threads. This thread priority is playing significant role when scheduler in kernel 2.6 assigns tasks into two queues: active and expired. Waked up thread is placed in active queue base on its priority. This means that when there are threads in systems with much different priorities, thread with bigger priority might be assigned again to active queue instead of expired queue. As long as there are threads in active queue as long threads from expiry queue will not get CPU time for execution. This might make situation while waked up threads stream can delay amount of time execution of tasks from expired queue. In the worst scenario this is possible even with only several CPU bound threads making situation in which low priority threads will be delayed more than several seconds.

In gateway example presented in Section 1 there is market driven possibility in which significant system applications are implemented to be executed in user space. As long as application providers are interested to not general public licence (GPL) it can not be implemented in kernel level. It is easy to imagine that set of user space applications can be executed in the system together with multiple kernel level tasks waked up quite often. Kernel priorities will take precedence over user space and will be serviced in active queue. In the same time expire queue threads will be still on hold.

The ATCA solutions on the market these days can give lots of communication opportunities to be used in professional systems. There is no communication connected issues any more. Separated parts of platform can exchange information base on standard backplane solutions offered by many suppliers. ATCA blades providers are proposing as well many systems working with energy save oriented CPUs like, e.g., ARM. For networking, these low performance cores are used to provide management opportunity for other CPUs like, e.g., network processors. Unfortunately, systems with good communication abilities might have some week points in low performance management core areas. If system design assumes existence of many Linux kernel space threads (often waked up) together with critical for system user space applications, so less priority threads might wait to be serviced even several seconds.

2.1. Time Demanding User Space Processes

Common practice made by ATCA system providers is system integration on the application level. As long as stable kernel with support is offered by companies like WindRiver or Montavista, as long management software can be offered by many other suppliers. Only in networking there exist many areas in which 3th party applications can be used.

For example SNMP stack or Internet key exchange (IKE) support can be purchased from protocol specialized suppliers and integrated together with blade interfaces. There is a big advantage for that kind of solution, especially for companies specialized in restricted areas like, e.g., signaling. Thanks to integration possibilities these companies can provide final systems to the market even without specialized knowledge in all system functionalities. All they need to provide is integration of solutions with support from application suppliers.

In the group of networking applications to be used with integration model there are some "time demanding" examples. Advanced telecommunication systems used in core networking are often designed to provide redundancy opportunities. For example in the case of gateway failure the system is prepared to switch over to backup gateway. This redundancy can be serviced by VRRP protocol (see Fig. 1).

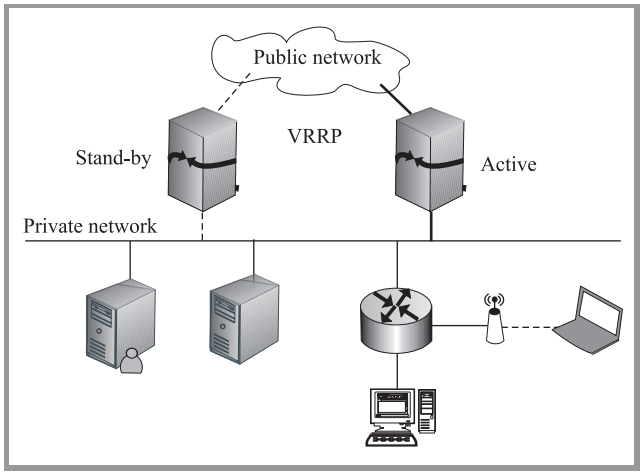

Fig. 1. Redundancy for network core nodes.

The VRRP protocol assumes continuous communication between active and backup gateway. In the case of communication lost for specified period of time, failover between gateways supposes to occur. In this VRRP example system is classified as unhealthy (dead), when packet exchange between gateways will not occur in given time.

One of the possible failover conditions would be user space VRRP application thread stocked in the expiry scheduler queue waiting until continuously waked up kernel threads will finally finish their jobs.

2.2. Critical Scenario Analyses

Linux scheduler is dealing with one run queue for each CPU in the system. Each run queue contains set of two priority arrays. All tasks begin in active priority array, and when they are executed on CPU there are moved to expired priority array and new time slice is calculated. Time slice describes time which given task will be able to spend on CPU before another task will be given a chance. A change between active and expiry priority array will take place when there will be no tasks in the first active array. Linux 2.6 scheduler is designed to schedule always all the tasks with the biggest priority (see Fig. 2). If there are couples of tasks with the same priority then they will be scheduled with round robin algorithm.

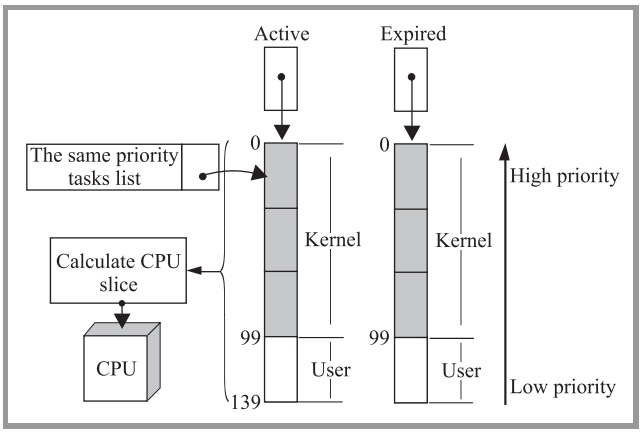

Fig. 2. Scheduler priority queue model for single CPU system.

In the Linux there can be user defined static values assigned to the priorities (*nice* from 20 to –19 by default 0). System is not intended to change static values to respect user input. To provide a difference between service I/O bound and CPU bound tasks scheduler uses dynamic priorities (0–139), which can award a bonus or depreciate task about 5 priority levels. Dynamic prioritization uses heuristic based on tracking how much time a task is sleeping against how long they are using CPU. Time T_{S_AV} is never intended to be bigger than $T_{\max}$ and a bonus to bigger priority is given to tasks with bigger T_{S_AV}. Priority can dynamically be changed based on average time T_{S_AV} of CPU waiting on CPU (I/O bound). When task is waked up after T_S to be executed on CPU then

$$\underset{T_{S\_AV}<T_{\max}}{\forall} T_{S\_AV} = T_{S\_AV} + T_S\,.$$

When task finishes using CPU after T_{CPU} then

$$\underset{T_{S\_AV}<T_{\max}}{\forall} T_{S\_AV} = T_{S\_AV} - T_{CPU}\,.$$

Scheduler will not perform any heuristic priority changes for real time tasks (see Fig. 2 – priorities 0–100). Real time tasks are always executed with the current priority. For the rest of tasks bonus B (maximum $B_{\max}$) will be calculated in the following way:

$$B = NTJ\left(\frac{T_{S\_AV}B_{\max}}{T_{\max}}\right),$$

where $NTJ - NS\_TO\_JIFFIES$ (see macro defined in sched.c [12]) depends on CPU frequency f [Hz],

$$NTJ(T) = \frac{T}{\dfrac{1000\,000\,000}{f\ [\mathrm{Hz}]}}\,.$$

When T_{S_AV} is high (I/O bound) then B might be 10 – task priority P will be increased about 5 and when T_{S_AV} is zero then B as well will be 0 – task priority P will be decreased about 5 levels.
Priority is an essential metric for scheduler to calculate time slice. The lowest dynamic priority process will get the biggest time slice T_{CPU} (for given $P_{\max}$ – maximal priority and $P_{\max\_U}$ – maximal user priority):

$$T_{CPU} = \max\left(T_{CPU\_DEF}\frac{(P_{\max}-P)}{\frac{P_{\max\_U}}{2}}, T_{CPU\_\min}\right),$$

$$T_{CPU\_DEF} = \frac{100\,f\ [\mathrm{Hz}]}{1000},$$

$$T_{CPU\_\min} = \max\left(\frac{5\,f\ [\mathrm{Hz}]}{1000}, 1\right).$$

Figure 3 presents a set of possible waiting for CPU times for 15 tasks with different priorities (CPU 800 MHz). Lower priorities tasks will receive less CPU time than task with bigger priorities. This chart presents data for single active queue without changing to expiry queue.

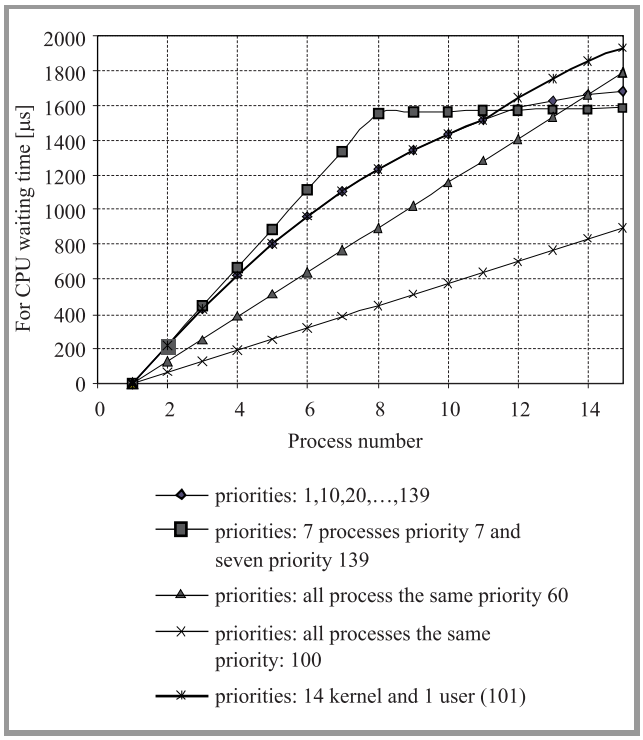

Fig. 3. Scheduler active queue tasks possible waiting for CPU time.

If system administrator assign the biggest possible nice priority to user space VRRP it is easy to prove that if there is many waked up processes in the active queue, then user space process will not be able to get CPU even after several milliseconds. It should be enough to set many processes in the kernel with high priorities. Delay in servicing VRRP process might be too big relative to its time demanding behavior. If network node will not send frame notification that he is alive there might be failover procedure started.

Task with higher priority will be given with longer CPU time than task with lower priority. The CPU time slice will even be longer on machines with higher CPU frequency (see Fig. 4).

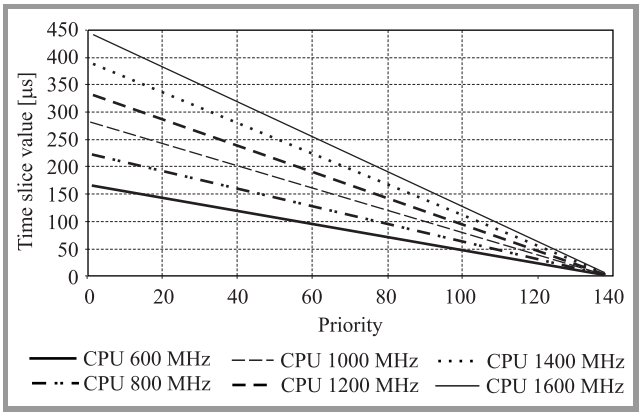

Fig. 4. Time slice estimation for different priorities on different CPUs.

To marginalize possibility of this situation there always can be said, that Linux kernel application supposes to be designed to avoid having important user space process stocked in the queue. In most cases it would be possible to work with design to make sure that database structures serviced in the kernel would have enough pointer references to avoid checking field by field. Unfortunately, close to this assumption exist many other possibilities (especially valid for ATCA) like, e.g., not enough memory to implement good enough data structures to avoid checking fields in the loop. Another common in the market reason is changing application assumptions when implementation is finished, that it is easier to find another solution to solve user space process stock issue than expensive redesign.
This set of explanations was accumulated in this section to assure about reasonability of researches presented in the next sections. Analyses of scheduler changes possibilities should always become first, before system designer decides to change base functionality of system kernel. Kernel level changes would make whole system less stable – unless validation in the field confirms that kernel patch works properly. On the basis of the following research results it should be much easier to decide if application should be redesigned or rather scheduler should be improved.

3. Scheduler Improvement for User Space Time Demanding Applications

Linux with its open source (OS) nature is giving this useful opportunity to provide changes even in the most critical parts of a code. This is allowed to change application as well as patch the kernel. Scheduler, as one of the most critical part of kernel, is already able to deal with user

processes base on heuristic method described in previous section.
To change a user space priority, there is average T_{S_AV} process sleeping time metric introduced in Linux kernel 2.6. This metric is working good to determine I/O bound threads. To have time demanding user space application running with bigger priority there is a different heuristic metric needed.

Scheduler metric to classify time demanding applications. To create a functional metric for user space process (which can be classified as time demanding) there needs to be such process characteristic introduced. Base on this characteristic the new metric can be introduced to classify task priority to be changed.
Time demanding user space process:

- is awaked periodically for a specified amount of time;
- in most cases required to deal with I/O peripherals;
- its awake time can be different – depends on process functionality.

Usage of I/O peripherals can match many other processes, not necessarily time demanding and is not a good characteristic for metric. Much more useful seems to be periodic activity of time demanding applications. If scheduler could classify that application requests CPU access every defined amount of time (different for different processes), it can reassign bigger priority to application process.
In order to be more flexible in scheduler changes it would be good to use variables already implemented in the kernel. To classify task as time demanding the average waiting for CPU time T_{S_AV} can not be easily used. For user space task this time depends on many conditions in kernel. For example T_{S_AV} can be completely different while kernel threads are requested to make big amount of calculations. Base on priorities kernel threads will be given with CPU time slice before user processes (see Fig. 3).
Opposite possibility to detect time demanding tasks would be eliminate these, which are not matching characteristic. Scheduler on the beginning could give the same big priority to all the processes to make sure that all time slices will be the same. To notify Linux scheduler that process/task should get a CPU (normally based on I/O) there is kernel variable need_resched = 1 used. If time demanding application would force need_resched = 1 periodically then T_{S_AV} should be enough to make task classification. There would be of course impact on whole system if scheduler would classify all tasks with the same priority at least for executing active, backup and again active queue. It should be enough to establish which process should be classified as time demanding. Unfortunately, this assumption could work when all of the processes would start the same time – which is bad assumption in the regular OS example.
Additional opportunity would be usage of average of T_{S_AV} to eliminate sporadic activity of bigger priority tasks activity. Scheduler could be easily changed in order to save in additional data structure K times T_{S_AV} when given PID is executed on CPU:

$$\underset{schedule(),PID}{\forall} T_{AV}(K) = T_{S\_AV}\,.$$

Arithmetic average could easily be calculated on the basis of the data collected in created table:

$$T_{AAV} = \frac{\sum_{N=1}^{K} T_{AV}(N)}{K}\,.$$

This method could be successful as long as K would be estimated correctly and K average time calculation would be repeated couple of times. Additionally two average values would never be the same and there the range of error would need to be considered here as well:

$$\begin{aligned}
&1_\text{estimation}: && T_{AAV}(1)\\
&2_\text{estimation}: && T_{AAV}(1) - X < T_{AAV}(2) < T_{AAV}(1) + X\\
&\cdots\cdots\\
&n_\text{estimation}: && T_{AAV}(1) - X < T_{AAV}(n) < T_{AAV}(1) + X\,.
\end{aligned}$$

If it would be enough to classify that task matches time demanding characteristic after $n = 2$ estimation, however probability that there is no mistake after $n = 3$ estimation would be much bigger.
Scheduler could be designed to increase process priority about A when 2_estimation = TRUE and about B when 3_estimation = TRUE ($B > A$).

4. Scheduler Improvement Measurement Results

Heuristic method in scheduler in kernel 2.6 assumes priority change about ± 5. It is not too much, especially when several active kernel space tasks exist in active and expired queues. Figures 5 and 6 present changes in waiting for CPU time for 15 user space tasks with priorities from 100 to 114 and 100, 103, …, 140.
Scheduler changes could provide more preemption than ± 5 change. Preemption patch [13] makes all tasks in-

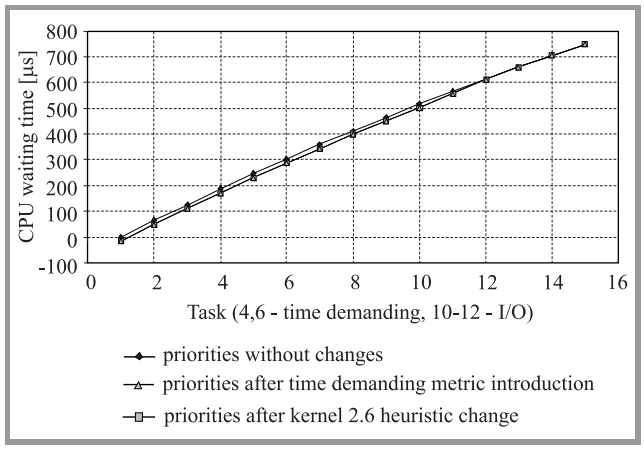

Fig. 5. Users space priority change – impact on waiting for CPU time – priorities 100–114.

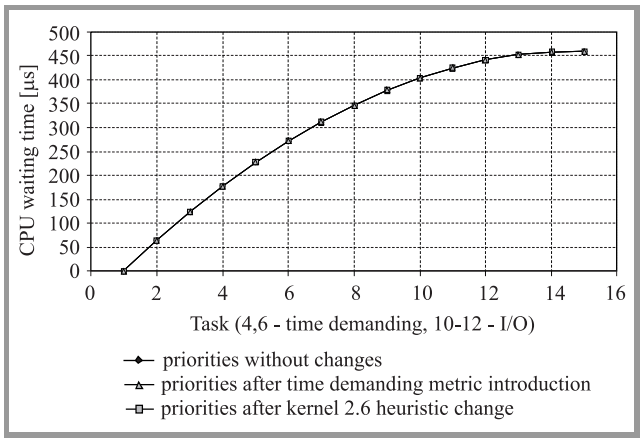

Fig. 6. Users space priority change – impact on waiting for CPU time – priorities 100, 103,. . ., 140.

cluding kernel soft-real-time available for priority change. For time demanding application executed in user space it would be more accurate to make opposite preemption and allow to change from user to kernel priority (from SCHED_NORMAL to SCHED_RR).

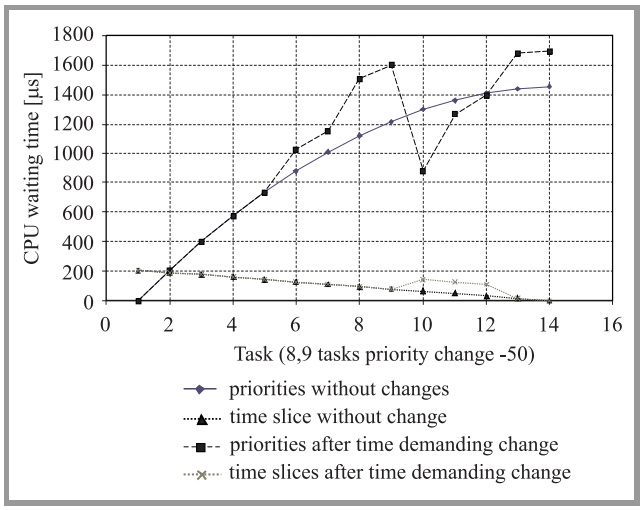

Fig. 7. Users space priority change – impact on waiting for CPU time and time slices – significant change –50.

Figure 7 presents total waiting for CPU and time slice values for priority change –50 (users pace moved to kernel).

Scheduler metric efficiency experiment. Time demanding processes classification metric bases on the average T_{S_AV} calculated after K measurement of T_{S_AV}. Average value is more valuable when it is calculated on the basis of more measurements. For scheduler it is not acceptable to make too many schedule() after priority change is done. For VRRP example if value K is determined incorrectly then scheduler could keep calculating which process should have priority changes while failover occurs. In the described metric method there is introduced value X which determines acceptable range to classify process request for CPU as periodic. Metric success basically depends on correct X value, which should be not too big (to not classify accidental tasks) and not too small (to catch periodic nature even if there is major change in the queue for bigger priorities).

Table 1
Time T_{AAV} for different number of measurements

K	User1(120)	User2(130)	User3(134)	User4(135)
1	1352	1384	1400	1409.6
2	1420	1452	1468	1477.6
3	**1431.466667**	**1463.46667**	**1479.46667**	**1489.06667**
4	1431.6	1463.6	1479.6	1489.2
5	**1420.48**	**1452.48**	**1468.48**	**1478.08**
6	1416	1448	1464	1473.6
7	1406.857143	1438.85714	1454.85714	1464.45714
8	1405.4	1437.4	1453.4	1463
9	1398.577778	1432.35556	1447.64444	1457.06667
10	1399.52	1431.52	1447.52	1457.12

Table 1 and Fig. 8 describe possible average T_{S_AV} calculated for different K. This example consider only active queue with 10 kernel space tasks (priority 0–99) and 4 user space tasks (priority 100–139). Every schedule() CPU is given to the next process from active queue for a time slice calculated on the basis of priority. For every one from 10 experiments user space tasks in active queue have the same priorities while kernel can change to simulate difference of kernel tasks in a given amount of time.

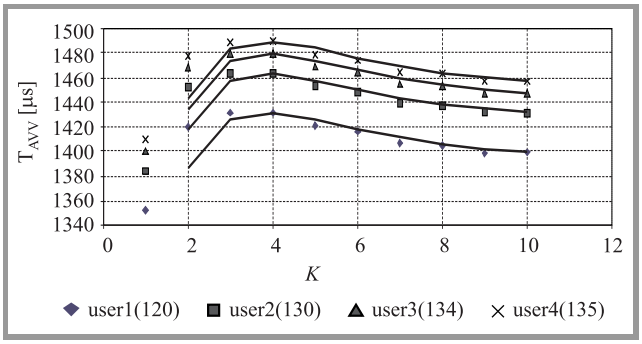

Fig. 8. Average T_{AVV} for user spaces processes and trend lines.

Measurement of $K = 10$ active queue can provide information about average error described in Table 2.
For this example $X = 80$ μs would be valuable for time demanding process metric and would classify processes much better than $X = 40$.
Scheduler should be as well resistant to an average calculation errors. For that it can elect a process to increase

Table 2
Difference between average measurement for $K = 10$

User	T_{AAV} (1)	max T_{AAV}	min T_{AAV}	X = max-min	x = (max-min)/2
User1(120)	1352	1431.6	1352	79.6	39.8
User2(130)	1384	1463.6	1384	79.6	39.8
User3(134)	1400	1479.6	1400	79.6	39.8
User4(135)	1409.6	1489.2	1409.6	79.6	39.8

```
X //average error range
C //allowed priority change for scheduler
PID //process ID
K //estimation
PRIO //process priority
TAVV[PID][K]

If schedule()
    K=K+1
    TAVV[PID][K] = TAVV
    //save average waiting time
    If (K==3)
         PRIO[PID]=PRIO[PID]+C
         //change priority
         Reschedule()
    end
    If (K==5)
         PRIO[PID]=PRIO[PID]+C*2
         //change priority
         Reschedule()
    end
    If (TAVV[PID][K] – TAVV[PID][K-1] > X)
         K=0
         //decline no time demanding processes
    end
    Reschedule()
end
```

Fig. 9. Scheduler estimation example pseudo code.

priority base on two values of K. Detailed algorithm is described in pseudo code on Fig. 9.

5. Summary

Time demanding user space application issue can be solved as many other computer science problems. To determinate if the cost of the solution is good enough to use it in the end user system a couple of numbers needs to be calculated together. Most important parts of the final grate would be the programming cost, improvement effect on real system, system stability after change.

Goal of this paper was to prove that such improvement in the kernel scheduler is possible and this or another idea can make time demanding user space application working more effective. According to measurement and calculation presented in previous section, Linux kernel scheduler can put more attention to the time demanding system activities. This can be done without breaking more important system rules. The scale of improvement depends on priority change level, which can be performed when process/task will be classified as time demanding. Presented solution shows as well that metric can depend on a set of additional parameters as classification range border or number of estimations. This leaves open door for system designers and developers and improvement, the base on several improvements can be parameterized for a dedicated system (e.g., ATCA SBC with a set of application running or ATCA line card with management application on it).

Acknowledgment

Effort sponsored by the Ministry of Science and Higher Education, Poland, under grant PBZ-MNiSW-02-II/2007.

References

[1] ATCA – PICMG 3.0 R2.0: ECN 3.0-2.0-001 [Online]. Available: http://www.picmg.org
[2] ATCA – Intelligent Platform Management Interface Specification Second Generation v2.0, Feb. 2006.
[3] L. Null and J. Lobur, *The Essentials of Computer Organization and Architecture*. Sudbury: Jones & Bartlett Publ., 2006.
[4] "Security Architecture for the Internet Protocol", RFC 4301.
[5] "Internet Key Exchange (IKEv2) Protocol", RFC 4306.
[6] Montavista Linux [Online]. Available: http://www.montavista.com
[7] WindRiver [Online]. Available: http://www.windriver.com
[8] Linux GPL [Online]. Available: http://www.gnu.org
[9] "Virtual Router Redundancy Protocol (VRRP)", RFC 3768.
[10] "A Simple Network Management Protocol (SNMP)", RFC 1157.
[11] J. Aas, "Understanding the Linux 2.6.8.1 CPU scheduler", SGI, 2005.
[12] Linux kernel sources [Online]. Available: http://kernel.org
[13] Linux kernel preemption project [Online]. Available: http://kpreempt.sourceforge.net/

Marcin Hasse received the M.Sc. degree in telecommunication from the Gdańsk University of Technology, Poland, in 2005. Currently he works for embedded computing leading company providing solutions for telecommunication market. His research interest and current work are related to operating system improvements for networking/telecommunication usage scenarios. He is an author of several publications in computer networking mechanisms improvements for end user services.
e-mail: marcin@hasse.pl
Gdańsk University of Technology
G. Narutowicza st 11/12
80-952 Gdańsk, Poland

Krzysztof Nowicki received his M.Sc. and Ph.D. degrees in electronics and telecommunications from the Faculty of Electronics at the Gdańsk University of Technology, Poland, in 1979 and 1988, respectively. He is an author or co-author of more than 100 scientific papers and an author and co-author of five books. His scientific and research interests include network architectures, analysis of communication systems, network security problems, modeling and performance analysis of cable and wireless communication systems, analysis and design of protocols for high speed LANs.
e-mail: krzysztof.nowicki@eti.pg.gda.pl
Gdańsk University of Technology
G. Narutowicza st 11/12
80-952 Gdańsk, Poland