Multi-threaded user and kernel-space library

Hristo Valchanov[1] and Simeon Andreev[2]

Abstract – **Developing of technology and the large range of possibilities offered by modern hardware allows the use of specialized high-performance approaches for the implementation of various software systems and algorithms. One of the most used and effective approach is the creation of multi-threaded software running in parallel on multiple processors. This paper presents the features of the implementation of multi-threaded library under Linux, which allows running both in user and kernel-space mode.**

Keywords – **Multi-threaded library, Threads, User-space, Kernel-space.**

I. INTRODUCTION

Development of technology and the large range of possibilities offered by modern hardware components allow the use of specialized high-performance approaches in the implementation of various software systems and algorithms. One of the most used and effective such approach is the creation of multi-threaded software running in parallel on multiple processors [7]. The actuality of this type of tasks and the need for new and improved modern applications argues mainly with the introduction of application oriented processors containing multiple independent cores (currently 6-8), which can operate independently of each other [2].

Every modern operating system supports multiprocessor operations. Important requirement for modern operating systems is the effective use of available hardware resources and provision them in the most appropriate way to user programs [1].

A current trend is using of multi-threaded libraries as an essential part of the most used and effective programming languages such as C ++, Java and others. The existence of different specific architectures makes this task difficult and sometimes even impossible for effective implementation. So there are many different standard libraries optimized for a particular architecture and operating system [5].

The paper presents the specifics of the implementation of multi-threaded library under Linux, which allows for operation both in user (user-space) and system (kernel-space) mode.

II. MOTIVATION

There are currently multiple implementations of multi-threaded libraries for different purposes. The GNU Portable Threads (Pth) [3] is a library created with the idea of a portable interface to a wide range of UNIX systems. The library Next Generation POSIX Threading (NGPT) [4] is developed by IBM for compatibility with POSIX standard for Symmetric Multi-Processing (SMP) machines. Linux Threads [8] is an implementation of the POSIX IEEE 1003.1c standard for Linux platforms. The library is based on the model one-to-one and operates in user mode. The new implementation of threads in Linux - Native Posix Thread Library (NPTL) [9] is also compatible with the POSIX standard, but it is based on a kernel-space model.

The implementations are optimized for machines with different architectures - uniprocessor, multiprocessor with shared memory, multiprocessor with distributed memory [6]. The modern processors are multi-core, which results in increased productivity. However, one should also take into account the fact that not every class of algorithms is subject to parallelism. There are algorithms for which has not yet found an effective multi-threaded implementation, or even inability to establish such one. On the other hand, there are many tasks for which the improvement in the speed of execution is great and justifies the additional difficulties that occur in multiprocessor operation. The software developer should have a choice of different functionality of multi-threaded libraries depending on the specifics of the various tasks to solve.

The presented library provides such flexibility. For this purpose a maximum identical user interface in both modes has been developed.

III. IMPLEMENTATION IN USER-SPACE MODE

A. Threads switching

The way of organizing threads switching by the dispatcher is essential for the effectiveness of multi-threaded library.

A possible approach for taking the processor from a user thread and providing it to the dispatcher is using an interrupt. In Linux the signals are convenient system for the realization of such user software interruptions. An interruption can be taken, for example, by a timer set to a specified interval. There are several problems making such a decision not very desirable. First, a very big advantage of the design of the entirely library in user-space, is the ability to fast switching between threads. The continuous use of signals and timers, however, requires switching to the kernel, which slows down repeatedly this usually fast operation. Second, storing of the user context that was interrupted by the timer is a difficult task. This is because when an interrupt function is executed, for some library functions it is not guaranteed to function properly.

Another possible approach is the dispatcher to work in a separate thread visible to the operating system, but this leads to other difficulties in implementation without providing the necessary effectiveness.

These are the main reasons why is using a non-preemptive dispatching in the proposed implementation of the user-space

[1]Hristo Valchanov is with the Department of Computer Science at Technical University of Varna, 1 Studentska Str., Varna 9010, Bulgaria, E-mail: hristo@tu-varna.bg.

[2]Simeon Andreev is with the Department of Computer Science at Technical University of Varna, 1 Studentska Str., Varna 9010, Bulgaria, E-mail: simeon.andreev90@gmail.com.

library. This solution also provides an opportunity to clearly show the main advantages and disadvantages of threads creating entirely in user-space mode.

B. Threads states in user-space mode

Each thread is represented by a special structure (Thread Control Block - TCB), containing the necessary information for its management. This information is used by the dispatcher for scheduling and context switching. The dispatcher takes care of the management of the events, which is very important for proper operation of the library. The dispatcher is called whenever a thread gives the CPU to another thread or when it locks automatically to an event. Figure 1 shows the states that a thread can take during its existence.

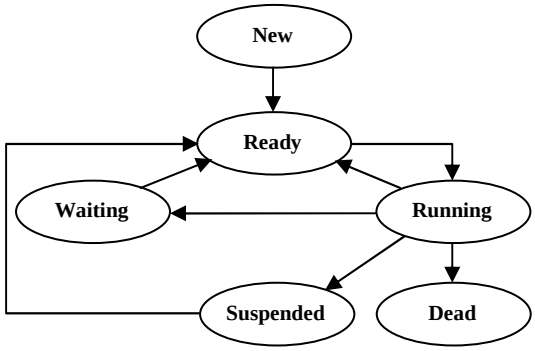

Fig.1. Threads states in user-space mode

Each of the states except *Running*, is represented with a priority queue of TCB of the threads which are in this state. A new created thread first is in the *New* state. From there, at the earliest opportunity the dispatcher can move it into the *Ready* state. On every dispatching, from the *Ready* queue is selected the most priority task which becomes *Running*. From the *Running* state a thread can go into any of the following states: *Dead* - if it finishes its execution, *Waiting* - if it is blocked to an event or into the *Ready* state otherwise. On every dispatching the threads in the *Waiting* queue are checked and moved to the *Ready* state if the appropriated events occur. In the *Suspended* state each thread can go if it is need not to be executed for a given period of time.

The non-blocking read/write operations, the thread sleeping for a specified time and the operations with synchronization primitives are entirely based on the event system.

IV. IMPLEMENTATION IN KERNEL-SPACE MODE

A. Threads states in kernel-space mode

The basic idea of this approach is to use the resources available to the operating system level for tasks dispatching. This allows for much better planning and using of resources of the entire system, but the disadvantage is that there is a slow switching between threads because of the need for system calls. The implementation of the threads in this mode is based on system call *clone()*. For the Linux kernel a thread is stored in the same structure, which is used for a separate process. Although that the basic information is stored in the structures of the operating system, it is still necessary to maintain data for the thread in the user context too. Such information, for example, is for the starting function and its argument. Maintaining this duplicated information allows a simpler implementation of some library functions.

Figure 2 shows a graph of the threads states in this mode.

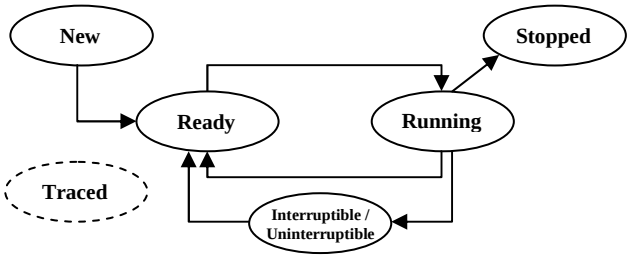

Fig.2. Threads states in kernel-space mode

The thread states are following:

- *Running* - The thread is either in the queue for starting or its execution is in progress.
- *Interruptible* - The thread is temporarily suspended while is awaiting the fulfillment of some condition. It wakes up on the event occurrence or on receipt of a signal.
- *Uninterruptible* – The state is identical with the preceding, the only difference is that the signals have no effect on the thread.
- *Traced* - When a thread is traced in the system (debugging mode).
- *Stopped* - When the execution of the thread is terminated. This state is achieved upon receipt of certain signals.

B. Threads synchronization in kernel-space

In the current library implementation are developed two types of synchronization primitives - *spinlocks* and *mutexes*. The spinlocks provide synchronization based on active waiting. Although this approach is not particularly effective, implementation is extremely simplified.

The second type primitives are based on waiting to release the processor. They are more complicated to implement because they require the use of system calls to the kernel. This is necessary for the dispatcher to suspend the execution of the current thread and to choose another thread to continue its work. In the proposed library the mutexes are implemented in the most effective way through a mixed approach. Initially, an attempt to short active waiting is made. If after that the mutex is still busy, the kernel blocks the thread execution. For this purpose is used a special tool for basic access locking - Fast User-Space Mutex (*futex*) [10].

In general, for the proper operation of a futex it is need to allocate a system semaphore. In the current implementation are used special (private) version of futex, which are local to the process, thus saves checking the semaphore internally in the kernel. This implies that the implementation is faster than that provided by the NPTL library where mutexes have a much more complex structure, slowing their use.

V. APPLICATION PROGRAMMING INTERFACE

The library provides an application programming interface that is maximum identical in both modes. The interface includes several groups of data types and library functions with different purpose such as:

- Types of data to work with system structures;
- Functions for initialization and completion of the work with the library;
- Functions for creating and destroying threads;
- Synchronization functions;
- Function for blocking and waiting;
- Functions for access to information for a separate thread.

VI. EXPERIMENTAL STUDY AND RESULTS

The testing of multi-threaded library was made with regard to the two basic types of system load:

- CPU bounded processes. These are processes that perform many and long time calculations.
- I/O bounded processes. These are processes that perform intensive system calls to the operating system. For example, saving in the files, waiting for an input from the user, using network communications.

The fastest execution for CPU bounded processes is expected to be achieved when the dispatcher uses a maximum possible period between switching of two threads. This allows saving of information and maximum use of processor caches. These processes almost always work without interruption, using all their time determined by the dispatcher. An optimal policy for them is to be allowed to work a long time, whereas it is not necessary their frequent starting.

On the other hand, the I/O bounded processes do not require long periods to use the CPU. This is because they rarely utilize it fully and often block alone. An optimal policy for them is the dispatcher to run them as much as possible more often for work.

Comparisons are made with the most common multithreading library *pthreads* and in particular with its most recent implementation in Linux - NPTL.

The experimental platform is based on Intel i7 2600K, running at 4.0GHz clock speed. The processor uses "HT" technology and can execute 8 threads simultaneously. Each core has its own cache memory: L1 - 64KB, L2 - 256KB and shared L3 cache size 8MB. The memory is DDR3 SDRAM - 8GB, 1600MHz, dual channel mode. The operating system is 64-bit Arch Linux distribution with kernel version 3.8 and the compiler - gcc 4.8.1 and glibc 2.17.

The first test involves matrix multiplication, as an example of a task comprising multiple calculations. There are created 5 separate threads, each of them multiplies 20 times a square matrices of size 100x100 elements. Figure 3 shows the obtained results.

As we expect, the *pthreads* and kernel-space implementations cope best with the test. The results are close, but the advantage is for *pthreads* library. From this test is shown the great disadvantage of user-space libraries - they can work on only one processor. The remaining two libraries take advantage of the available 8 cores. The test excludes blocking of threads.

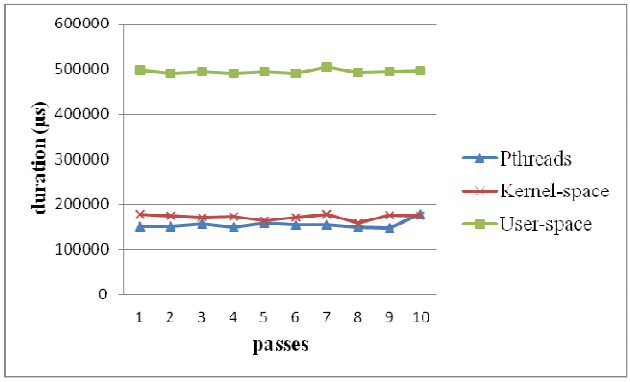

Fig.3. Results for matrix multiplication

The second test uses handling of input/output operations. There are created 2 threads, one reads from a file, and the other saves data in it. The pairs read/write operations are performed 1000000 times. It is expected that during the execution of some of the operations blocks could occur. Figure 4 shows the obtained results.

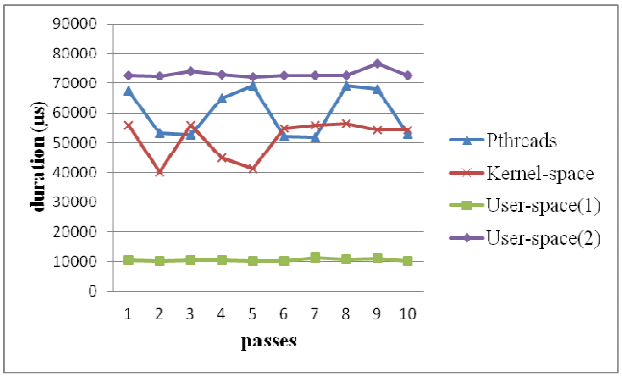

Fig.4. Results for I/O operations

The user-space library is used in two ways. With "1" is noted the use of non-blocking primitives provided by the current implementation, and with "2" - when they are replaced with standard *write()* and *read()*. As can be seen, the developed tools have better efficiency in comparison with the standard ones in Linux.

The *pthreads* and kernel-space libraries show the same performance with a slight preponderance of the kernel-space implementation.

The test results show also a greater fluctuation between different starts because it is more difficult to predict variants of execution.

The third test aims to assess the effectiveness of synchronization primitives. In the cycle of 1000000 iterations a mutex is locked and unlocked. There are created 5 separate threads, each of which performs the described action. Figure 5 shows the obtained results.

It is evident that the implementation of mutex in *pthreads* library is slower and requires an average 615154µs.

The version in the kernel-space library is about 1.5 times faster, thanks to a simple and effective structure of organization of the threads. Also the advantage is due to the fact that an active waiting is used instead immediately suspending of the execution of the thread.

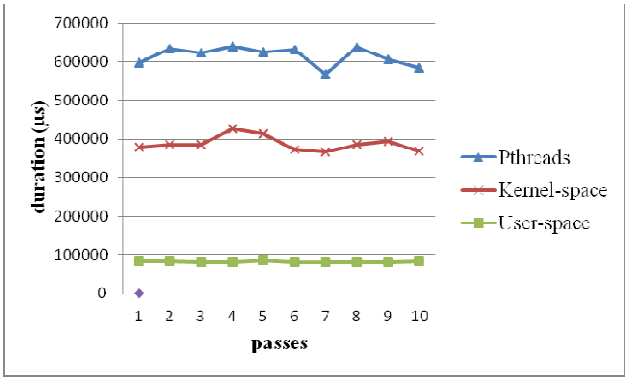

Fig.5. Results for synchronizing primitives

Quite expectedly the user-space implementation is proved fastest with an average of 82875µs. This is due to the fact that slow system calls are not used.

VII. CONCLUSIONS

This paper presents an implementation of a multithreaded library under Linux, functioning both in user-space and kernel-space mode. Some features of implementation are given. Experimental comparisons and evaluations with the famous library *pthreads* have been made.

The results show that for presented library it is achieved identically and in specific cases high performance. The developed library is relatively small, which results in faster compilation. The code is written in a simple way, which allows the use of the library for learning multithreaded programming.

Goal of future work is to improve the signal processing in user-space mode, as well as developing additional synchronization primitives, such as conditional variables and monitors.

REFERENCES

[1] Haldar S., A. Aravind. Operating Systems. Dorling Kindersley Ltd. 2010.
[2] Herlihy M, N. Shavit. The Art of Multiprocessor Programming. Morgan Kaufmann, 2008.
[3] Engelschall R. Portable Multithreading. The signal stack trick for user-space thread creation. In Proceedings of 2000 USENIX Annual Technical Conference, 2000, pp.18-23.
[4] IBM Corporation. Next Generation POSIX Threading, November 2002.
[5] Ljumovic M. C++ Multithreading Cookbook. Packt Publishing, 2014.
[6] Love R. Linux System Programming. O'Reilly. 2013.
[7] Sandem B. Design of multithreaded software. Wiley Inc. 2010.
[8] Leroy X. The Linux Threads Library. http://pauillac.inria.fr/~xleroy/.
[9] Native Posix Thread Library. http://people.redhat.com/drepper/nptl-design.pdf.
[10] Futex Semantics and Usage. http://man7.org/linux/man-pages/man7/ futex.7.html.