Supplementary Notebook (RTEP - Brazilian academic journal, ISSN 2316-1493)

TASK MANAGER FOR GENERAL-PURPOSE OPERATING SYSTEMS

Alexey I. Martyshkin [1]

[1]*Candidate of technical sciences, docent, associate Professor of sub-department «Computers and systems», Penza state technological University. (440039, Russia, Penza, Baydukov Proyezd / Gagarin Street, 1a/11, e-mail: alexey314@yandex.ru).*

Abstract: This paper suggests a possible task manager implementation for general-purpose operating systems. The aim of the study is to maximize processor utilization in exclusive mode for general-purpose operating systems. The subject research field of this study is relevant today in the light of global informatization and the urgent issue of improving computing performance. In order to achieve the objectives, set in this study, several particular problems have been solved, which include analysing various scheduling strategies for computational task execution, comparing specifics of various operating systems, choosing a system resource allocating method, developing a software for running calculations in priority mode, and minimizing scheduling and dispatching overheads. During the study, we have analysed four families of general-purpose operating systems and their features. A summary information on the utilized scheduling and dispatching algorithms has been prepared based on the analysis results, which allowed choosing the appropriate approach to solve the problem under consideration. A resource allocation method has been chosen and implemented as software unit, based on mechanisms common to the considered systems that developed software starts computations under a configuration corresponding to the system's topology, sets a real-time scheduling policy for threads, assigns them to available computational cores, and independently dispatches task execution. The effectiveness of the developed software is confirmed by test runs and measurement of such indicators as runtime, number of context switches and accesses to external memory. The main results obtained can be applied when designing new and improving the existing general-purpose operating systems.

Keywords: scheduling algorithm, computations, time slicing, overheads, operating system, optimization, task scheduling and dispatching, task manager, priority, performance, processor, resource.

INTRODUCTION

Conceptual provisions on the essence of audit the today's world sets high requirements to the computing performance. Task optimization with the execution time reduced from one minute to 59 seconds will most likely go unnoticed by a regular PC user. The situation is different when large amounts of data shall be processed (16). Reducing the execution time of a continuously repeated task from 60 to 59 seconds allows finishing the entire number of operations 1.67% faster or perform 1.67% more operations in the same time period. Thus, about six days of runtime per year can be gained. It is not always possible to optimize the computations, in particular (17). Task runtimes can be divided into several stages: including initialization, CPU wait, waiting for other resources, CPU execution, termination. Initialization includes creating the data structures required for the operating system's kernel, loading the application image into the main memory, opening the required files, allocating other resources, and initial CPU execution scheduling. Tasks waiting for CPU in a ready tasks queue is a standard procedure in multi-tasking systems with time-sharing. Typically, the CPU executes a priority task while the rest are waiting in a queue. The operating system's kernel uses dynamic priorities preventing any task to monopolize the CPU (34). Task execution by the CPU at some point in time means its code is executed at that moment by the CPU. If a certain event is necessary for the further execution of a task, the task is added to the queue and wait for this event. Such an event may be I/O being finished without CPU participation, system resource being freed (for example, a synchronization primitive), and etc.

I/O waiting time depends on the load and bandwidth of the I/O subsystem. CPU execution time depends on the program code optimization. The queue waiting time for ready tasks depends on their number in the system, the required computing power, the ratio of priorities and configuration of OS scheduler and task manager. Initialization time almost completely depends on the operating system implementation specifics (34). When studying the subject research field, we analysed literature sources (5; 18; 11; 12; 20) and (18; 11; 12; 20; 19; 12; 11; 2; 38) in order to find any poorly studied issues. Some important issues related to implementation of scheduling and dispatching subsystems of computer systems did not find the necessary coverage in published works, however, they are mentioned in the following sources (27; 37; 15; 28;) and (31; 4; 3; 7). This study discusses the problems associated with task execution scheduling and dispatching. The objective for this study is to maximize CPU resources utilization in exclusive mode for general-purpose operating systems. The subject research field considered in this study is relevant today due to global informatization and the urgent issue of improving computing performance. To achieve the objective defined above, the study solves the following particular problems: analyse various strategies for computational task execution scheduling; compare the specifics of various operating systems; choose a resource allocation method; develop a software for running calculations in priority mode to minimize scheduling and dispatching overheads; perform comparative performance measurements.

General-purpose operating systems are used in personal computers and server systems, usually for data processing (computing, processing, storage). The main requirement to these systems is wide support for various software packages (34). Low cost of personal computers and a wide range of solved tasks contributed to their widespread use (35; 33), as well as induced rapid development of time-sharing

operating systems. Such systems are intended to solve the following problems: ensure full utilization of computer resources; ensure that each task is executed by the CPU; ensure each task receives the required amount of computing resources depending on their priority. Time-sharing systems use CPU time slicing for task execution. In the general case, the task is interrupted as a time slice expires, and the next task is selected for execution. Implementation details depend on the algorithm used for task scheduling.

SCHEDULING ALGORITHMS

In the context of this study, it makes sense to define scheduling algorithms that have an independent implementation or are a part of a more complex algorithm: First Come First Served (FCFS, FIFO) — execution in the order of arrival (39). The first ready task is executed by the CPU, each subsequent task enters the queue and waits for the completion of all tasks ahead of it. This ensures execution in case none of the tasks occupy the CPU forever. This algorithm is not related to time-sharing since tasks are executed sequentially without interrupting. Round Robin (RR) — tasks are performed cyclically in fixed time intervals (32). At the end of each interval, the task is added to the end of the queue. For each task, this process is repeated until it is completed. The cyclic nature of the algorithm ensure that each task takes 1/N of CPU time, where N is the total number of tasks. Shortest Job First — each time a task is selected for execution with the smallest execution time available in the queue (32). Though, for this algorithm to work, the time required to complete a certain task shall be known. Fixed Priority Pre-emptive — at any given time, the CPU executes the highest priority task (36). When a task with the highest priority appears, the task being executed is interrupted and is queued, freeing the resources for the priority one. Execution of low priority tasks is not guaranteed. Earliest Deadline First — each task is assigned a deadline, exceeding which makes further execution no longer required (9).

Priority Queue — a queue with task processing order not depending on the time when tasks are added to the queue (32). Each time, a task with the highest priority is selected for execution from the entire queue. With this case, two implementations are possible: a pre-emptive algorithm, when a task in the queue with a priority higher than the one being processed, pre-empts the latter, and a non-pre-emptive algorithm, when a new task with the highest priority waits for the task already executed by the CPU to be completed. Multilevel queue — tasks are added to one of the queues that differ in priorities and assigned to it until completion (32). A task is selected for execution from the priority queue. Queues can implement various scheduling algorithms. For example, two queues can be used for different classes of tasks, a FCFS queue with a higher priority, and a lower priority queue with a priority-based internal implementation. Tasks from the first queue can pre-empt tasks of the second queue from the CPU. Thus, two classes of tasks can be simultaneously processed — tasks of ordinary and higher priority, which is pre-emptive. Multilevel feedback queue — unlike a multilevel queue, tasks can switch from one queue to another, usually after exhausting their CPU quota (32).

OVERHEADS AND THEIR IMPACT

Overheads consist of the time required for scheduling/dispatching tasks and related processes. The scheduler's runtime depends on the computational complexity of its implementation and, generally, the number of tasks in the system. Typically,

dispatching is well optimized and takes very little CPU time, however, the related context switching process entails cache cooling, TLB invalidation and, in the worst case, page swapping. Cache cooling means losing cache entries and frequent cache misses before the cache warms up. A cache miss is followed by an access to a lower, slower memory level. TLB contains addresses of virtual pages of the process. TLB invalidation results in double number of accesses to RAM (8). Process page swapping to disk is a way to deal with low memory. Process pages not currently required are saved to disk, and the necessary pages are loaded into RAM. This leads to an active exchange with the slowest memory in a PC. Computing performance massively decreases in the case of active swapping (34).

The following operating systems have been selected to study the functionality and utilized scheduling/dispatching algorithms: FreeBSD 10; GNU\Linux 4; Oracle Solaris 11; Microsoft Windows 7. The choice of operating systems for analysis is dictated by their prevalence in computing applications, as well as by an intention to cover various operating system families to identify the most suitable implementations of the required functionality. This study uses the concept of a "task" to denote an independent dispatch unit. Using the terminology of Windows, Solaris, and FreeBSD operating systems, a "process" is a system entity containing the environment necessary for application execution, including program code. A "thread" is an entity that stores the execution state of a sequence of instructions related to a process. One process can have one or several threads (26; 1; 10). All Linux threads are actually processing that share common system resources (14). As these resources are already allocated, thread creation overheads are reduced. The result is an entity, which is dispatched as threads in other operating systems. Thus, all considered operating systems use threads as a minimum dispatch unit.

TASK SCHEDULERS

The FreeBSD kernel scheduler uses several priority executions queues (run queues, Figure 1a): itqueues, rtqueues, queues, idqueues — in decreasing order by priority (26).

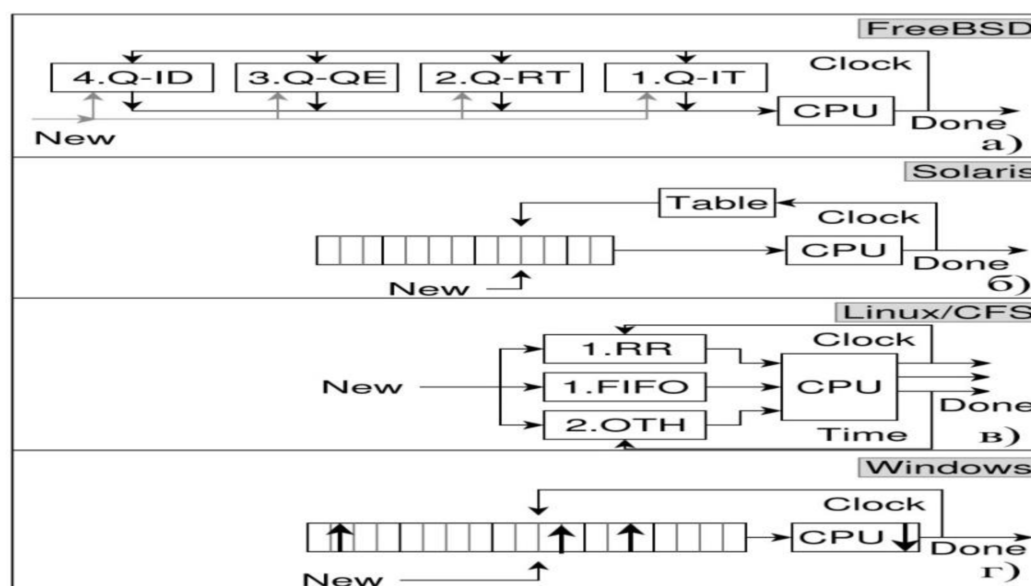

Figure 1. Schedulers of various operating systems

A priority task is selected from the non-empty higher priority queue. Task execution is forcibly interrupted after the time slice allocated to it is elapsed, and its dynamic priority is recounted. The task is added to the end of a queue containing tasks of equal priority. A task that locked a shared resource receives the priority of a task waiting for this resource, if it is higher. The Solaris OS scheduler uses multiple queues as per the number of priorities, as well as a priority conversion table (Figure 1b). When the time slice allocated for a task expires, the priority of this task is decreased, however, at the same time, its next time slice is increased. On the contrary, the task, which frees the CPU before the time slice expires, gets a higher priority and a smaller time slice. As a result, the waiting time of small frequent tasks is reduced, they get executed faster, while large computational tasks are executed more efficiently because their execution is interrupted less often. Tasks that lock system resources receive a higher priority, which reduces the execution time of such tasks, reduces the time till resource release, and reduces waiting time of tasks waiting for this resource.

CFS scheduler, included in the main branch of Linux kernel development, uses several different data structures (Figure 1c) (6). Deadline class tasks have a clearly defined execution time and deadline. They are scheduled for execution by the Earliest Deadline First algorithm. Real-time tasks have static priorities and are scheduled according to one of the following algorithms: First Come First Served or Round Robin. Fixed time slices are used for the latter case, after which the tasks are added to the end of the queue according to its priority. In the first case, the higher priority task is executed completely, without interrupting. For standard priority tasks, CFS implements a weighted fair queue algorithm. Being assigned to the CPU, the task receives execution time proportional to the queue waiting time and inversely proportional to the number of tasks ready for execution.

Windows OS family task scheduler uses several classes and priority levels: low, medium, high, and two intermediate levels (Figure 1d). Each task is assigned a base priority by the scheduler. A task with the highest priority is always selected for execution. The kernel scheduler can increase its priority in one of the following four cases: 1. an event occurs, for example, user input waiting to be handled by a task; 2. the task is performed interactively in the foreground; 3. the task is unlocked; 4. tasks ready for execution are randomly and periodically assigned a higher priority. As a CPU time slice expires, the dynamic priority of a task decreases by one, down to lower limit. Thus, the scheduler fights the "starvation" effect by increasing the priorities of waiting tasks and bringing them back to normal as this effect decreases. Interactive processes are also assigned higher priorities.

PRIORITY SCHEDULING STRATEGIES

FreeBSD, Linux, and Solaris operating systems support Pthreads, a POSIX-compatible library that provides a single interface for multi-threaded applications. Solaris also provides its own Solaris Threads interface, introduced before Pthreads. Solaris Threads provides such features as parallelism management, thread sleep/wakeup, read/write locks mechanism, and daemon thread creation. In contrast, it does not support thread cancellation and POSIX scheduling strategies. Windows has its own threads implementation — Windows Threads. This interface has many features that are not supported in POSIX, such as: per user scheduling, fibers manually managed in the scope of a stream, critical sections as a synchronization primitive. Table 1 contains a summary on the interfaces for interaction with threads, schedulers, and their

configurations.

Table 1. Interfaces of operating systems

OS family	FreeBSD	GNU/Linux	Solaris	Windows
Thread management	POSIX.1 Pthreads			Windows Threads
Scheduler interface	sched_setscheduler (SCHED_FIFO)			SetPriorityClass (REAL TIME_PRIORITY_ CLASS)
CPU affinity	pthread_ setaffinity_ np	sched_ setaffinity	processor_ bind	SetThreadAffinityMask

As can be seen from the table, FreeBSD, Linux, Solaris OSs support POSIX standardized interfaces (29), and therefore the code that implements interactions via these interfaces will be easily portable between systems. The POSIX standard does not include the CPU affinity functionality for threads, however, all the considered operating systems have similar capabilities, although without a cross-compatibility.

SOFTWARE IMPLEMENTATION

Computing process scheme

The computing process is based on scheme that uses the general principles implemented in the operating systems considered above. To maximize CPU utilization by computational tasks and minimize the number of context switches, SCHED_FIFO scheduling strategy has been selected. This allows to bind the required number of threads to the CPU, which are not pre-empted after time slices expire. Tasks are implemented as functions contained in a queue being part of the application. Task managing includes saving the result returned by the function, reading and moving the pointer, and calling a new function. To exclude possible thread migrations between processors, an explicit processor binding is used. Software interrupts are also prohibited.

Definition of system topology

System topology, the number of processors, physical and logical cores are determined at application start. On Linux, this information is extracted from sysfs unified virtual file system, which serves as an interface for kernel structures. /sys/devices/system/cpu/ virtual folder contains cpuN sub-folders with files describing the topology and state of the corresponding logical kernel number N:

```
int topology (int ** cpus)
{
struct dirent ** cpusdir; int c, count;
count = scandir (SYSFS_CPU, &cpusdir, cpu_dir_filter, alphasort);
if ((*cpus = malloc (sizeof(int) * count)) == NULL)
{
err (EXIT_FAILURE, "can't allocate memory for " \ "cpus array");
}
for (c=0; c < count; ++c)
{
assert (1 == sscanf (cpusdir[c]->d_name, "cpu%d", &(*cpus)[c]));
}
```

```
return count;
}
```

After the topology is determined, a decision is on the CPU affinity of the computing threads:

```
cpu_set_t set;
/* ... */
if (tharg -> cpu >= 0) /* if < 0, do not bind */
{
CPU_SET (tharg -> cpu, &set);
if ( -1 == sched_setaffinity ( 0, sizeof(set), &set ) )
{
err (EXIT_FAILURE, "Can't set affinity mask");
}
}
```

Assigning real-time priority

In order to arrange the computational process, we decided to create a dedicated software that prepares threads for computation and initiates tasks. Each thread runs on a dedicated core/processor, and the number of threads is equal to the number of cores minus one. One core is allocated to the needs of various system and user tasks. This allows to achieve the maximum degree of parallelism, while maintaining system responsiveness. Since CPU time-sharing strategy does not allow monopolizing the cores for practical computations, we decided to assign the computational tasks to the real-time class, having the highest priority in the system. Real-time tasks have the highest priority, which does not change depending on the execution time, and are never pre-empted by low-priority tasks. Computational tasks get maximum CPU time.

FCFS task scheduling algorithm has been selected since this algorithm results in minimal overheads and is easy to implement. Time slicing is not used, and the task is not forcibly interrupted until its execution is finished. No unnecessary context switches are used, the caches are warm most of the time and work efficiently.

SCHED_FIFO scheduling strategy is set usingsched_setscheduler system output, implementation of the call in the source code is shown below.

```
if (-1 == sched_setscheduler (0       /* self */, SCHED_FIFO,
&((struct sched_param) { .sched_priority = 99 })) )
{
warn ("Can't set SCHED_FIFO policy");
}
```

Interrupt binding

Interrupt handling can be assigned to a specific core. A proc virtual file system interface is used for this purpose. /proc/irq/M/smp_affinity file (where M is the interrupt number) contains a mask, high bits of which define permissions for being handled by the core with the corresponding number. At start-up, the developed software records the mask and the corresponding free core. For example, mask 8 corresponds to core 3.

Application task queue arrangement

The application queue contains pointers to functions and arguments passed to them. Computational threads independently select tasks by copying pointers to a function and an argument into their local context and moving the current queue element's pointer. Mutex synchronization primitive is used for sharing. The block diagram of the task selection algorithm is shown in Figure 2.

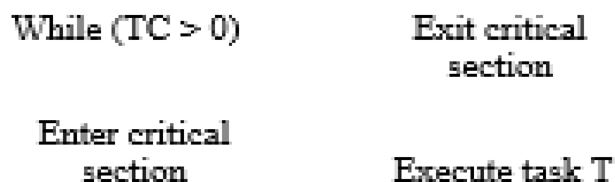

Figure 2. Queued task selection algorithm

THE RESULTS OF THE CONCLUDED RESEARCH
Measurements of the software performance indicators

The measurements of counters built into the CPU and Linux operating system kernel (starting from version 2.6.31) have been made using the perf utility (Mikheev et al., 2016; Martyshkin et al., 2019; Martyshkin & Martens-Atyushev, 2019). The number of context switches, TLB misses, cache misses of the first and last level, the number of thread migrations from processor to processor, and the total execution time have been measured. Test run and measurements have been carried out on a PC with an Intel®Core™ i5 6200U processor, running GNU/Linux OS (x64, kernel version 4.4.0-79-generic). CPU specifications:

- Number of physical cores: 2.
- Number of logical cores: 2.
- Operating frequency: 2.30 GHz.
- L1 instruction cache: 32 KB per core.
- L1 data cache: 32 KB per core.
- L2 cache: 256 KB per core.
- L3 cache: 3 MB per processor.

The operating system's kernel has been allocated 2 GB of RAM. As a useful load, a task has been chosen that implements a cyclic array traversal with a varying step and increasing element values. The array size was 5 MB. The external load has been modelled by four processes actively performing operations with memory (allocated a total of 1.5 GB of RAM). The measurement results are summarized in Table 2 as the average values of collections with confidence intervals for each of 100 independent measurements. The values of all indicators are expressed in the number of events, except for time, which is expressed in seconds. The first column of the table contains a list of measured indicators (branches, cache-misses, etc.) and explanations regarding the test environment:

- No load — no CPU consuming tasks have been running on the machine in addition to the test task. CPU load before the test launch remained at 3-5%, which indicated idle state of the CPU.
- Load 1 — CPU load simulating application has been also running on each core of the machine in addition to the test task. This application worked with a small amount of RAM (0.1% of the total amount of RAM), constantly performing arithmetic operations.
- Load 2 — CPU load simulating application has been also running on each core of the machine in addition to the test task. Each instance of this application worked with a large amount of RAM (about 19% of the total amount), constantly performing arithmetic operations.

The remaining column headers contain three-character codes. The first letter denotes the application configuration (D is the default configuration by OS, C is the configuration prepared by the developed application). The number corresponds to the number of threads (T) started by the application.

Table 2. Measurement results

No load	**D1T**	**C1T**
branches	1.436e+10 ± 4.664e+06	1.435e+10 ± 1.030e+07
cache-misses	1.495e+09 ± 2.677e+06	1.502e+09 ± 8.676e+05
cache-references	2.697e+09 ± 2.879e+06	2.705e+09 ± 1.098e+06
dTLB-load-misses	478 ± 55	104 ± 24
dTLB-loads	6.458e+10 ± 1.078e+07	6.459e+10 ± 2.472e+07
dTLB-store-misses	797 ± 210	12.4 ± 2.1
dTLB-stores	2.157e+10 ± 3.665e+06	2.159e+10 ± 7.813e+06
page-faults	1.087e+03 ± 1.074e-01	1.088e+03 ± 1.640e-01
context-switches	48.6 ± 2.9	1.20 ± 0.09
cpu-migrations	0.80 ± 0.09	1.00 ± 0.00
time	22.1 ± 0.1	21.4 ± 0.0
No load	**D2T**	**C2T**
branches	1.435e+10 ± 5.726e+06	1.431e+10 ± 1.864e+06
cache-misses	1.700e+09 ± 2.195e+06	1.663e+09 ± 8.744e+05
cache-references	2.717e+09 ± 2.457e+06	2.693e+09 ± 1.334e+06
dTLB-load-misses	2.276e+04 ± 2.956e+03	503 ± 41
dTLB-loads	6.448e+10 ± 1.706e+07	6.459e+10 ± 9.164e+06
dTLB-store-misses	2.297e+05 ± 4.241e+04	44.8 ± 5.5
dTLB-stores	2.155e+10 ± 2.234e+06	2.157e+10 ± 4.342e+06
page-faults	1.098e+03 ± 1.640e-01	1.099e+03 ± 1.074e-01
context-switches	2.191e+03 ± 1.429e+01	3.20 ± 0.21
cpu-migrations	1.00 ± 0.00	3.40 ± 0.18
time	11.8 ± 0.0	11.7 ± 0.0
No load	**D3T**	**C3T**
branches	1.434e+10 ± 5.941e+06	1.429e+10 ± 2.760e+06
cache-misses	1.759e+09 ± 2.635e+06	1.769e+09 ± 2.020e+06
cache-references	2.726e+09 ± 7.285e+05	2.721e+09 ± 2.193e+06
dTLB-load-misses	4.783e+03 ± 1.447e+02	2.780e+03 ± 1.627e+02
dTLB-loads	6.459e+10 ± 5.919e+06	6.458e+10 ± 1.162e+07
dTLB-store-misses	1.110e+03 ± 7.133e+01	226 ± 18
dTLB-stores	2.158e+10 ± 6.639e+06	2.153e+10 ± 3.971e+06
page-faults	1.107e+03 ± 8.765e-02	1.108e+03 ± 1.640e-01

context-switches	675 ± 68	7.80 ± 0.56
cpu-migrations	13.0 ± 1.7	6.80 ± 0.21
time	10.6 ± 0.0	10.8 ± 0.0
No load	**D4T**	**C4T**
branches	1.435e+10 ± 2.488e+06	1.432e+10 ± 3.669e+06
cache-misses	1.845e+09 ± 1.930e+06	1.799e+09 ± 2.599e+06
cache-references	2.747e+09 ± 1.440e+06	2.741e+09 ± 2.915e+06
dTLB-load-misses	2.086e+04 ± 1.252e+03	1.621e+03 ± 1.599e+02
dTLB-loads	6.453e+10 ± 1.124e+07	6.454e+10 ± 1.465e+07
dTLB-store-misses	5.382e+03 ± 2.319e+02	82.8 ± 8.2
dTLB-stores	2.158e+10 ± 6.639e+06	2.154e+10 ± 3.601e+06
page-faults	1.117e+03 ± 8.765e-02	1.118e+03 ± 2.235e-01
context-switches	6.570e+03 ± 9.412e+01	12.2 ± 0.7
cpu-migrations	10.2 ± 1.1	8.00 ± 0.31
time	10.0 ± 0.0	11.2 ± 0.1

Table 2. continued.

Load 1	**D1T**	**C1T**
branches	1.435e+10 ± 7.193e+06	1.436e+10 ± 1.761e+06
cache-misses	1.513e+09 ± 8.936e+05	1.484e+09 ± 1.705e+06
cache-references	2.733e+09 ± 3.375e+06	2.737e+09 ± 5.586e+05
dTLB-load-misses	8.761e+04 ± 1.947e+03	66.0 ± 9.9
dTLB-loads	6.457e+10 ± 2.873e+07	6.444e+10 ± 5.903e+06
dTLB-store-misses	1.030e+06 ± 2.830e+04	3.50 ± 1.16
dTLB-stores	2.157e+10 ± 8.985e+06	2.154e+10 ± 3.123e+06
page-faults	1.088e+03 ± 8.765e-02	1.088e+03 ± 1.323e-01
context-switches	2.988e+03 ± 7.981e+01	2.40 ± 0.10
cpu-migrations	27.0 ± 1.3	1.00 ± 0.00
time	34.0 ± 0.2	23.1 ± 0.0
Load 1	**D2T**	**C2T**
branches	1.435e+10 ± 6.432e+06	1.430e+10 ± 1.702e+06
cache-misses	1.702e+09 ± 2.208e+06	1.663e+09 ± 3.761e+05
cache-references	2.717e+09 ± 1.496e+06	2.696e+09 ± 6.666e+05
dTLB-load-misses	7.739e+04 ± 1.574e+03	469 ± 27
dTLB-loads	6.448e+10 ± 2.065e+07	6.455e+10 ± 2.640e+06
dTLB-store-misses	8.214e+05 ± 2.236e+04	82.6 ± 22.5
dTLB-stores	2.155e+10 ± 6.607e+06	2.155e+10 ± 7.560e+05
page-faults	1.097e+03 ± 8.765e-02	1.099e+03 ± 2.556e-01
context-switches	4.220e+03 ± 5.518e+00	3.20 ± 0.09
cpu-migrations	1.40 ± 0.11	1.00 ± 0.00
time	23.7 ± 0.0	11.5 ± 0.0
Load 1	**D3T**	**C3T**
branches	1.434e+10 ± 6.226e+06	1.431e+10 ± 3.532e+06
cache-misses	1.782e+09 ± 2.505e+06	1.771e+09 ± 8.000e+05
cache-references	2.757e+09 ± 4.269e+06	2.742e+09 ± 1.417e+06
dTLB-load-misses	1.282e+04 ± 5.318e+02	4.382e+03 ± 2.493e+02
dTLB-loads	6.449e+10 ± 1.561e+07	6.456e+10 ± 1.069e+07
dTLB-store-misses	4.292e+03 ± 2.184e+02	398 ± 14
dTLB-stores	2.155e+10 ± 9.231e+06	2.154e+10 ± 4.791e+06
page-faults	1.108e+03 ± 8.765e-02	1.109e+03 ± 1.074e-01
context-switches	3.133e+03 ± 3.443e+01	3.80 ± 0.16
cpu-migrations	108 ± 2	2.80 ± 0.16

time	21.1 ± 0.1	12.6 ± 0.0
Load 1	**D4T**	**C4T**
branches	1.433e+10 ± 7.305e+06	1.433e+10 ± 3.099e+06
cache-misses	1.814e+09 ± 1.722e+06	1.797e+09 ± 2.400e+06
cache-references	2.753e+09 ± 4.099e+06	2.748e+09 ± 6.591e+05
dTLB-load-misses	1.247e+04 ± 3.250e+02	2.194e+03 ± 2.202e+02
dTLB-loads	6.455e+10 ± 1.287e+07	6.449e+10 ± 7.020e+06
dTLB-store-misses	4.281e+03 ± 1.577e+02	174 ± 13
dTLB-stores	2.154e+10 ± 6.842e+06	2.155e+10 ± 4.983e+06
page-faults	1.118e+03 ± 1.640e-01	1.118e+03 ± 1.074e-01
context-switches	3.684e+03 ± 3.068e+01	4.60 ± 0.53
cpu-migrations	60.6 ± 3.8	4.20 ± 0.35
time	18.9 ± 0.0	11.3 ± 0.1

Table 2. continued.

Load 2	**D1T**	**C1T**
branches	1.435e+10 ± 5.606e+06	1.436e+10 ± 2.663e+06
cache-misses	1.606e+09 ± 1.047e+06	1.579e+09 ± 8.044e+05
cache-references	2.719e+09 ± 2.412e+06	2.739e+09 ± 1.995e+06
dTLB-load-misses	3.959e+05 ± 1.314e+05	1.530e+06 ± 2.442e+03
dTLB-loads	6.465e+10 ± 1.587e+07	6.459e+10 ± 1.663e+07
dTLB-store-misses	4.462e+06 ± 1.462e+06	1.839e+07 ± 2.129e+04
dTLB-stores	2.158e+10 ± 4.435e+06	2.157e+10 ± 3.151e+06
page-faults	1.396e+03 ± 1.344e+02	2.624e+03 ± 8.765e-02
context-switches	3.245e+03 ± 5.795e+01	2.20 ± 0.09
cpu-migrations	32.8 ± 1.1	1.00 ± 0.00
time	35.1 ± 0.2	23.4 ± 0.0
Load 2	**D2T**	**C2T**
branches	1.431e+10 ± 6.404e+06	1.434e+10 ± 3.165e+06
cache-misses	1.749e+09 ± 2.172e+06	1.693e+09 ± 1.645e+06
cache-references	2.712e+09 ± 3.882e+06	2.710e+09 ± 2.503e+06
dTLB-load-misses	1.233e+06 ± 3.981e+04	1.162e+06 ± 8.296e+04
dTLB-loads	6.453e+10 ± 3.515e+07	6.466e+10 ± 9.244e+06
dTLB-store-misses	1.346e+07 ± 4.751e+05	1.310e+07 ± 1.051e+06
dTLB-stores	2.156e+10 ± 6.402e+06	2.160e+10 ± 2.077e+06
page-faults	4.166e+03 ± 1.753e-01	4.371e+03 ± 5.479e+01
context-switches	4.207e+03 ± 7.115e+00	3.20 ± 0.16
cpu-migrations	2.60 ± 0.41	1.40 ± 0.11
time	24.0 ± 0.0	11.9 ± 0.0
Load 2	**D3T**	**C3T**
branches	1.435e+10 ± 2.764e+06	1.433e+10 ± 4.876e+06
cache-misses	1.823e+09 ± 3.568e+06	1.797e+09 ± 2.812e+06
cache-references	2.766e+09 ± 4.407e+06	2.738e+09 ± 1.993e+06
dTLB-load-misses	9.419e+04 ± 2.005e+04	7.027e+04 ± 1.814e+04
dTLB-loads	6.456e+10 ± 2.016e+07	6.457e+10 ± 1.806e+07
dTLB-store-misses	2.084e+04 ± 5.140e+03	1.922e+03 ± 5.547e+02
dTLB-stores	2.155e+10 ± 4.381e+06	2.156e+10 ± 6.855e+06
page-faults	2.233e+03 ± 3.841e+02	2.949e+03 ± 5.516e+02
context-switches	2.981e+03 ± 6.360e+01	5.20 ± 0.32
cpu-migrations	119 ± 2	3.20 ± 0.35
time	21.1 ± 0.1	12.6 ± 0.0
Load 2	**D4T**	**C4T**

branches	1.434e+10 ± 2.563e+06	1.432e+10 ± 2.607e+06
cache-misses	1.869e+09 ± 5.291e+06	1.810e+09 ± 1.661e+06
cache-references	2.761e+09 ± 2.695e+06	2.748e+09 ± 1.722e+06
dTLB-load-misses	7.792e+04 ± 4.837e+03	6.143e+03 ± 1.514e+03
dTLB-loads	6.454e+10 ± 1.784e+07	6.445e+10 ± 9.041e+06
dTLB-store-misses	1.428e+04 ± 4.364e+02	844 ± 233
dTLB-stores	2.151e+10 ± 8.464e+06	2.153e+10 ± 4.089e+06
page-faults	2.448e+03 ± 5.472e+01	1.326e+03 ± 9.031e+01
context-switches	3.707e+03 ± 3.763e+01	7.60 ± 0.41
cpu-migrations	75.0 ± 4.5	5.40 ± 0.26
time	19.6 ± 0.1	11.7 ± 0.1

Table 2 shows that computation time with the developed application is virtually not dependent on the system load, created by other applications. Negative effects such as CPU migration during execution, and context switches are minimized. dTLB-load-misses and dTLB-store-misses have been reduced by an average of 1.45% compared to the initial values (compared with D*T configurations). Cache misses are virtually unchanged throughout the test, which is apparently due to the test tasks implementation specifics — the cyclic traversal algorithm for an array of several megabytes in size with a step significantly larger than unity, which prevents loading this array into the CPU cache. Four families of general-purpose operating systems and their features have been analysed during the study. Analysis results allowed to prepare a summary of the algorithms used for task scheduling and dispatching. The following general principles have been identified: Utilization of pre-emptive multi-tasking with CPU time slicing; Computing resources are provided depending on the execution history; Separately scheduled classes of real-time tasks; Ability to assign tasks to individual cores; Ability to bind interrupts to individual cores. This allowed deciding on the approach used to solve the considered problem. A resource allocation method has been chosen and implemented as software unit, based on mechanisms common to the considered systems. The proposed application starts the computations in the configuration corresponding to the system topology, sets the real-time scheduling policy for threads, assigns them to the available cores, and independently schedules task execution. The effectiveness of the developed software is confirmed by test runs and measurement of such indicators as runtime, number of context switches and accesses to external memory.

ACKNOWLEDGMENT

The article is published with the support of the scholarship from the President of Russian Federation to young scientists and graduate students in 2018-2020 (SP-68.2018.5).

REFERENCES

1. Baldwin, J.H. (2002). *Locking in the Multithreaded FreeBSD Kernel*. BSDCon, 27-35.

2. Barban, A.P., Ignatushenko, V.V., & Podshivalova, I.Yu. (2003). On the effectiveness of dispatching methods for complex sets of tasks in heterogeneous multiprocessor computing systems. *Automation and Telemechanics, 10,* 66-79.

3. Biktashev, R.A., & Vashkevich, N.P. (2017). *Structural implementation and modelling of scheduling/ dispatching algorithms in parallel computing systems based on non-deterministic automata logic.* New information technologies and systems: Collection of scientific articles of the XIV International Scientific and Technical Conference dedicated to the 70th anniversary of the Computer Engineering Department and the 30th anniversary of the Computer-aided Design Systems Department, 12-17.

4. Biktashev, R.A., Vashkevich, N.P., & Kiselev, S.V. (2016). *Development of a process scheduling tool for multiprocessor systems.* Modern technologies in science and education - STNO-2016: Proceedings of the international scientific-technical and scientific-methodical conference: in 4 volumes. Ryazan State Radio Engineering University; Under general editorship of O.V. Milovzorov, 304-308.

5. Bogatyrev, V.A., & Golubev, I.Yu. (2013). Optimal dispatching in distributed computing systems with nodes grouped in clusters. *Bulletin of computer and information technologies, 8*(110), 36-40.

6. Bovet, D., & Cesati, M. (2007). *LINUX kernel, 3 ed.* – BHV-Petersburg.

7. Chasovskikh, Ye.G. (2017). *Dispatching criteria for heterogeneous distributed computing systems*. Intelligent Information Systems Proceedings of the All-Russian Conference with International Participation, 97-99.

8. Chen, J.B., & Bershad, B.N. (1994). The impact of operating system structure on memory system performance. *ACM SIGOPS Operating Systems Review. – ACM, 27*(5), 120-133.

9. Chiussi, F.M., & Sivaraman, V. (2003). Guaranteeing data transfer delays in data packet networks using earliest deadline first packet schedulers: USA patent 6532213.

10. Cohen, A., & Woodring, M. (1998). *Win32 Multithreaded Programming.* – Oreilly & Associates Incorporated.

11. Demyanyuk, D.A., & Kogan, D.I. (2013). Issues of informational and algorithmic support for uniprocessor task dispatching. Bulletin of Moscow State University of Instrument Engineering and Computer Science. *Series: Instrument engineering and computer science, 44,* 83-90.

12. Dokuchaev, A.N. (2012). On the evaluation of the effectiveness of dispatch mechanisms of real-time multiprocessor systems, taking into account long-term locking effects. *Software Engineering, 9,* P. 2-7.

13. Dokuchaev, A.N. (2012). Specific aspects of ultralight tasks dispatching in multiprocessor real-time computing systems. *Information Technologies, 2*, 14-18.

14. Drepper, U., & Molnar, I. (2003). The native POSIX thread library for Linux. White Paper, Red Hat Inc.

15. Egorov, V.Yu. (2011). Criteria for assessing task scheduling effectiveness for a multiprocessor operating system. *Software Engineering, 3,* 29-33.

16. Filippenko, P.N., Shashelov, A.A., & Seitova, S.V. (2010). Creating systems for big data computations: problems and trends. *Proceedings of the Southern Federal University. Technical sciences, 113*, 12.

17. Gergel, V.P. (2010). *High-performance computing for multiprocessor multicore systems.* Moscow: Moscow University Press, P. 534.

18. Khludova, M.V. (2010). *Operating systems. Process scheduling and dispatching: textbook*. - St. Petersburg: Publishing House of the Polytechnic University, - 83 p.

19. Kurnosov, M.G., & Paznikov, A.A. (2012). Decentralized scheduling algorithms for spatially distributed computing systems. *Tomsk State University Bulletin. Management, computer engineering and computer science, 1*(18),133-142.

20. Lupin, S.A., Than Zo Woo, Zhuo Myu Htun. (2012). Using random search algorithms to solve the dispatching problem in distributed service systems. *News of higher educational institutions. Electronics, 3*(95), 40-46.

21. Martyshkin, A.I. (2013). Mathematical modelling of task managers for multiprocessor computing systems on the basis of open-loop queueing networks: Abstract of Ph.D. thesis in Engineering: 05.13.18. Penza State Technological University. Penza, - 23 p.

22. Martyshkin, A.I. (2016). Mathematical Modelling of Tasks Managers with the Strategy of Separation in Space with a Homogeneous and Heterogeneous Input Flow and Finite Queue. *ARPN Journal of Engineering and Applied Sciences, 11*(19), 11325-11332.

23. Martyshkin, A.I., & Martens-Atyushev, D.S. (2019). Mathematical modelling and evaluation of the characteristics of specialized reconfigurable systems based on a common bus at the stage of synthesis of the system configuration. *Journal of Advanced Research in Dynamical and Control Systems, 11*(8 Special Issue), 2852-2860.

24. Martyshkin, A.I., & Yasarevskaya, O.N. (2015). Mathematical modelling of Task Managers for Multiprocessor systems on the basis of open-loop queuing networks. *ARPN Journal of Engineering and Applied Sciences, 10*(16), 6744–6749.

25. Martyshkin, A.I., Pashchenko, D.V., & Trokoz, D.A. (2019). *Queueing Theory to Describe Adaptive Mathematical Models of Computational Systems with Resource Virtualization and Model Verification by Similarly Configured Virtual Server*. (2019). Proceedings. International Russian Automation Conference, RusAutoCon 2019. no. 8867620.

26. McDougall, R., & Mauro, J. (2006). *Solaris internals: Solaris 10 and OpenSolaris kernel architecture*. – Pearson Education.

27. McKusick, M. K., Bostic, K., Karels, M. J., & Quarterman, J. S. (1996). *The design and implementation of the 4.4 BSD operating system* (Vol. 2). Reading, MA: Addison-Wesley.

28. Mikheev, M.Yu., Zhashkova, T.V., Meshcheryakova, E.N., Gudkov, K.V., & Grishko, A.K. (2016). *Imitation modelling for the subsystem of identification and structuring data of signal sensors.* Proceedings of 2016 IEEE East-West Design and Test Symposium, EWDTS 2016. 7807748.

29. Roganov, V.R., Roganova, E.V., Micheev, M.J., Zhashkova, T.V., Kuvshinova, O.A., & Gushchin, S.M. (2018). Flight simulator information support. *Defence S and T Technical Bulletin, 11*(1), 90-98.

30. Semenov, A.O. (2015). *A stochastic approach to queue dispatching.* Optoelectronic devices and devices in pattern recognition, image and symbol information processing

systems: Proceedings of the XII International Scientific and Technical Conference, 331-333.

31. Semenov, A.O. (2016). *Cyclic queue dispatching algorithms modelling specifics and their implementation in the Simulink system.* New information technologies and systems: Collection of scientific articles of the XIII International Scientific and Technical Conference, 179-181.

32. Silberschatz, A. et al. (1998). *Operating system concepts.* – Reading: Addison-Wesley, – V. 4.

33. Stallings, W. (2002). *Operating Systems: Translated from English 4th ed. Moscow: Publishing house "Williams*". - 848 p.

34. Tanenbaum, A., & Bos, H. (2015). *Modern operating systems.* 4th ed. - St. Petersburg: Peter, 1120 p.

35. Tanenbaum, A., & Woodhull, A. (2007). *Operating systems: design and implementation.* 3rd ed. - St. Petersburg: Peter, - 704 p.

36. Unice, W.K. (2004). *Deterministic preemption points in operating system execution*: USA patent 6802024.

37. Vashkevich, N.P., Biktashev, R.A., & Kiselev, S.V. (2014). Task scheduling algorithm verification for a multiprocessor system using Stateflow tools. *Challenging issues of the humanities and natural sciences, 12-1*, 42-46.

38. Voevodin, V. V., Voevodin, Vl. V. (2002). *Parallel computing.* St. Petersburg: BHV-Petersburg, - 608 p.

39. Zhao, W., & Stankovic, J.A. (1989). Performance analysis of FCFS and improved FCFS scheduling algorithms for dynamic real-time computer systems. Real Time Systems Symposium, 1989. Proceedings. – IEEE, 156-165.