*Journal of* ELECTRICAL ENGINEERING, VOL. 52, NO. 3-4, 2001, 74–80

# PERFORMANCE AND TUNING OF THE UNIX OPERATING SYSTEM

**Milan Tichý** [*] — **Petr Zemánek** [**]

This article defines basic measures for the UNIX operating system performance. We present methods for performance monitoring, interpretating the results of measurement, and performance tuning. Methods for increasing the performance of processes are discussed. An example of measuring and finding optimal values of the kernel tunable parameters is presented.

K e y w o r d s: Operating system, process, file system, performance tuning.

## 1 INTRODUCTION

The problem of the performance of computer systems is rather complex, requiring good knowledge of computer architecture, UNIX design, and performance-monitoring tools.

When a computer system is slow, it is very difficult to get the most out of any given configuration. Many different factors play a role in determining a system's response. Usage patterns, I/O configuration, CPU configuration, and software configuration all contribute to a system's behavior.

Above all, optimizing system performance is a matter of making tradeoffs. It is not possible to optimize one aspect of performance without compromising some other aspect.

System performance can be affected by a number of things that generally fall into the following categories [8]:

- *Hardware:* Is the CPU fast enough and are there adequate peripherals to accomplish the task at hand?
- *Operating system:* Is the system configured properly for the current environment?
- *Application software:* Are applications designed for efficient processing and ease of use?
- *People:* Are people trained on the system and applications to optimize their productivity?
- *Changes over time:* What changes in workload and user requirements can be expected to occur?

How these factors interact together can have a critical impact on system performance.

A system performs poorly because one or more of its resources are being taxed too heavily. When resources cannot keep up with demand, user applications are forced to wait for them, resulting in a longer execution time and a slower response. Finding these bottlenecks is the most important step in performance tuning.

Once one or more system bottlenecks are found, it is necessary to be able to identify *why* the shortages have occurred. UNIX monitoring tools are fairly adequate in helping to figure out which resource is being taxed too heavily. But they give little information as to the cause of overutilization.

In looking at improving a system performance, we can opt to improve either the system throughput, the user response time, or both [16]. *Throughput* is the amount of work the system is performing as a whole. *Response time*, on the other hand, is a measure of how long it takes to finish some task.

Although these terms are often used interchangeably, there is a big difference between them. If we have hundred users, increasing the throughput means that all hundred users *collectively* get more work done. It does not indicate that every one of these users experiences a faster response. As a matter of fact, the response time could have slowed for some but speeded up for others.

## 2 COMPONENTS CONTRIBUTING TO A PROCESS RUNNING TIME

Many different components contribute to a program's total running time. Here is a summary of the different components [9]:

- **User-state CPU time:** The actual amount of time the CPU spends running the user's program in the user state. It includes the time spent executing library functions but excludes the time spent executing system calls (*ie*, time spent in the UNIX kernel on behalf of the process). User-state time is under the programmer's control.

[*] Academy of Sciences of the Czech Republic, Institute of Information Theory and Automation, Department of Computer Systems, Pod vodárenskou věží 4, 182 08 Prague 8, Czech Republic, E-mail: tichy@utia.cas.cz

[**] Czech Technical University in Prague, Faculty of Electrical Engineering, Department of Computer Science and Engineering, Karlovo nám. 13, 121 35 Prague 2, Czech Republic, E-mail: zemanekp@sun.felk.cvut.cz

ISSN 1335-3632 

- **System-state CPU time:** The amount of time the CPU spends in the system state (*ie*, the amount of time executing kernel code) on behalf of the program. This includes the time spent executing system calls and performing administrative functions on the program's behalf. The distinction between the time spent in simple library routines and the time spent in system services is important and often confused.
- **I/O time:** The amount of time the I/O subsystem spends servicing the I/O requests that the job issues. Under UNIX, I/O time is difficult to measure; however, there are some tools for determining whether the I/O system is overloaded and some configuration considerations that can help alleviate load problems.
- **Network time:** The amount of time that the I/O subsystem spends servicing network requests that the job issues. This is really a subcategory of I/O time and depends critically on configuration and usage issues.
- **Time spent running other programs:** As the system load increases, the CPU spends less time working on any given job, thus increasing the elapsed time required to run the job. This is an annoyance, but barring some problem with I/O or virtual memory performance, there is little we can do about it.
- **Virtual memory performance:** This is by far the most complex aspect of system performance. Ideally, all active jobs would remain in the system's physical memory at all times. In practice this is impossible, even on systems with large memory configurations. When physical memory is fully occupied, the operating system starts moving parts of jobs to the disk, thus freeing memory for jobs it wants to run. This takes time. It also takes time when these disk-bound jobs need to run again and therefore need to be moved back into memory. When running jobs with extremely large memory requirements, system performance can degrade significantly.

## 3 KERNEL CONFIGURATION

The *kernel* [1,13] is the heart of the UNIX operating system. It is the program that manages memory, schedules processes, manages I/O, and does all of the other low-level tasks that make the system work. The kernel is not a "process", like the programs we run from the keyboard. It pre-exists all processes and is responsible for starting the first process. It is the life-support system that allows everything else to work.

Because it is so important, the kernel has some key privileges. On most UNIX systems, it is *always* resident in the processor's physical memory.[1] Other programs can be swapped or paged to disk, but not the kernel. Therefore, the kernel should be as small and compact as possible. But like most software, the UNIX kernel grew over time: new features and compatibility modes were added, bug-fixes were larger than the code they replaced, and so on. As a result, the kernel and its tables can require a significant portion of the system's total memory, particularly on small systems. One important reason to create a custom kernel is to make it as small as possible: trim away unneeded features, and make kernel tables as small as comfortably possible.

## 4 MEASURED RESULTS

We used SGI INDY workstation and SGI IRIX 6.2 operating system for measuring. The configuration of the workstation (*hinv* command [5]) was as follows:

```
Iris Audio Processor: version A2 revision 4.1.0
1 100 MHZ IP22 Processor
FPU: MIPS R4600 Floating Point Coprocessor Revision: 2.0
CPU: MIPS R4600 Processor Chip Revision: 2.0
On-board serial ports: 2
On-board bi-directional parallel port
Data cache size: 16 Kbytes
Instruction cache size: 16 Kbytes
Main memory size: 160 Mbytes
Vino video: unit 0, revision 0, IndyCam connected
Integral ISDN: Basic Rate Interface unit 0, revision 1.0
Integral Ethernet: ec0, version 1
Integral SCSI controller 0: Version WD33C93B, revision D
  CDROM: unit 5 on SCSI controller 0
  Disk drive: unit 4 on SCSI controller 0
  Disk drive: unit 2 on SCSI controller 0
Graphics board: Indy 8-bit
```

We paid attention to three kinds of the specific regions of system performance tuning. First, we explored CPU capacity and scheduling. Next, we explored memory management (*ie*, paging and swapping) and I/O (specially disk) management.

We designed a specific set of programs for each kind of measuring for this purpose.

We want to present here an example of our measuring, *ie* watching system performance while changing kernel tunable parameters *slice_size* and *nbuf*.

We must say that the workstation we used for measuring was not entirely dedicated to measurement purposes only. It is also used as mail and WWW server. So, we have to consider some measuring inaccuracies that were caused by unbalanced load of the workstation during performing the measurements. This kind of measuring inaccuracies is considered in the Section 4.1.

### 4.1 CPU Capacity and Scheduling — Parameter *slice_size*

In this part, we will focus on the kernel tunable parameter **slice_size** [6]. This parameter is the default process time slice, expressed as a number of ticks of the system clock. The frequency of the system clock is expressed by the constant $Hz$, which has a value of 100. Thus each unit of *slice_size* corresponds to 10 milliseconds. When a process is given control of the CPU, the kernel lets it run for *slice_size* ticks. When the time slice expires or when

[1] The only exception are systems that have the MACH kernel. As far as I know, these are some new versions of OSF.

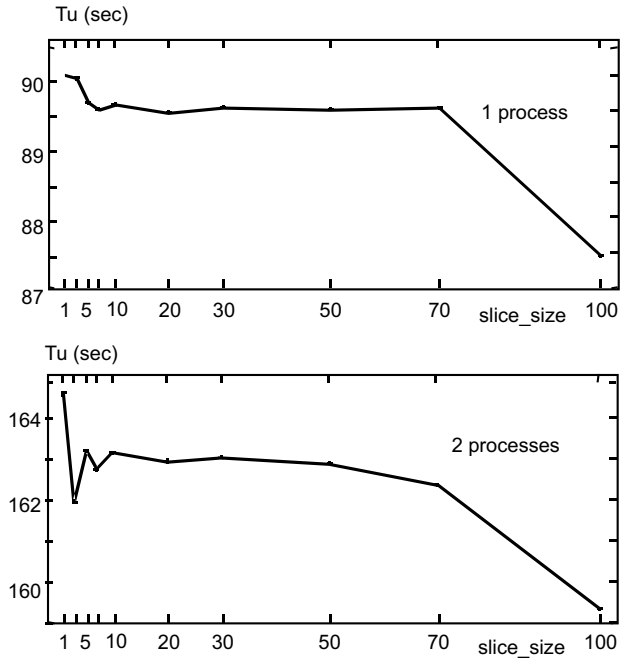


**Fig. 1.** User time as a function of *slice_size*

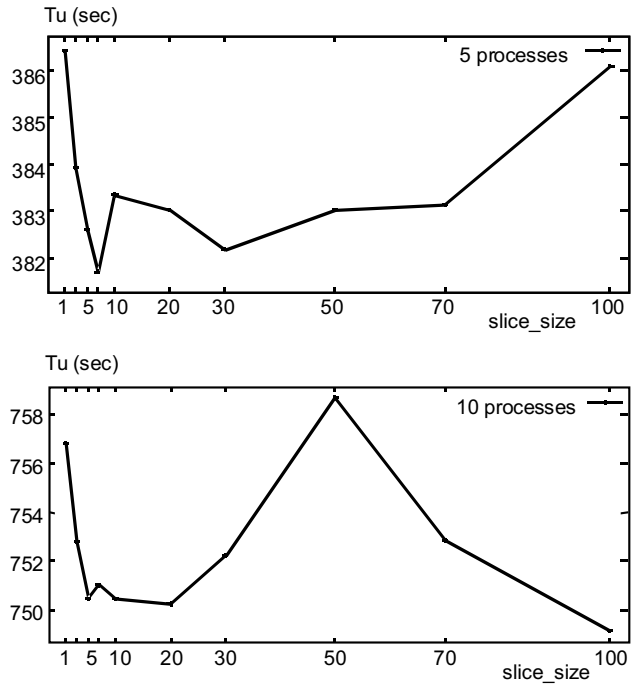


**Fig. 2.** User time as a function of *slice_size*

the process voluntarily gives up the CPU (for example, by calling *pause* or by doing some other system call that causes the process to wait), the kernel examines the run queue and selects the process with the highest priority that is eligible to run on that CPU.

Since *slice_size* is an integer, the default value is 3. This means that the default process time slice is 30 milliseconds. The *slice_size* parameter is defined in `/var/sysgen/mtune/disp`.

We prepared a CPU-bound program for the purpose of this measuring. We spawned the specified number of copies of this program so that they ran concurrently. In this way, we spawned 1, 2, 5, 10, 15, and 20 copies of it.

We changed the *slice_size* parameter values within the range 1–100. We kept watch over the operating system response in these values: 1, 3, 5, 7, 10, 20, 30, 50, 70, and 100.

We were interested in the following cumulative activity counters of the operating system, in the case of *slice_size* parameter:

- *user time* — the time that a process spent in the user mode.
- *runq-sz* — run queue of processes in memory and runnable.

It enabled us to find out the following relations:

1. User time $T_u$ as a function of *slice_size* parameter for different number of processes, $T_u(slice\_size)$. These relations are depicted in Figures 1–3. The essential question is why we paid attention to the user time $T_u$. We wanted to monitor CPU capacity and scheduling. So, we prepared tasks that oppress the CPU extremely and do not require much of the operating system servicing. These processes require, as much as possible, CPU time to be assigned to them. They spend most of their time in the user mode. The time that these processes spend in the kernel mode is negligible.

First, we will look at these relations generally. We can notice that the values of $T_u$ as a function of *slice_size* are decreasing rapidly within the *slice_size* parameter range of 1–3(5). This is dependent on the number of concurrently-running CPU-bound processes. If more processes are running, the difference is more distinct within this interval. Further, we can see that the fluctuation of $T_u$ is less distinct within the range of 3(5)–100. The values of user time $T_u$ decreases globally within this interval, if we increase the number of concurrently-running processes. It is clear if we look at Figures 1 and 3. The exceptions in Figure 2 were caused by measuring inaccuracy.

Now, we attempt to find the most appropriate value of the *slice_size* parameter so that this parameter setting should be favorable for all cases, *ie* for the situation we plan to spawn one CPU-bound process as well as more CPU-bound processes.

If we compare all cases and consider the situation when the operating system seems to be stable, we can find out this situation comes up for the values within the *slice_size* parameter range of 3–5. Lower values are absolutely unacceptable from the point of view of the user time $T_u$. Higher values are not favorable due to instability of the values of $T_u$ when we spawn the different number of the CPU-bound processes. We have to respect other results too as we will see.

2. Run queue of processes in memory and runnable *runq-sz* as a function of *slice_size* parameter, $runq\text{-}sz(slice\_size)$ (see Figures 4–6).

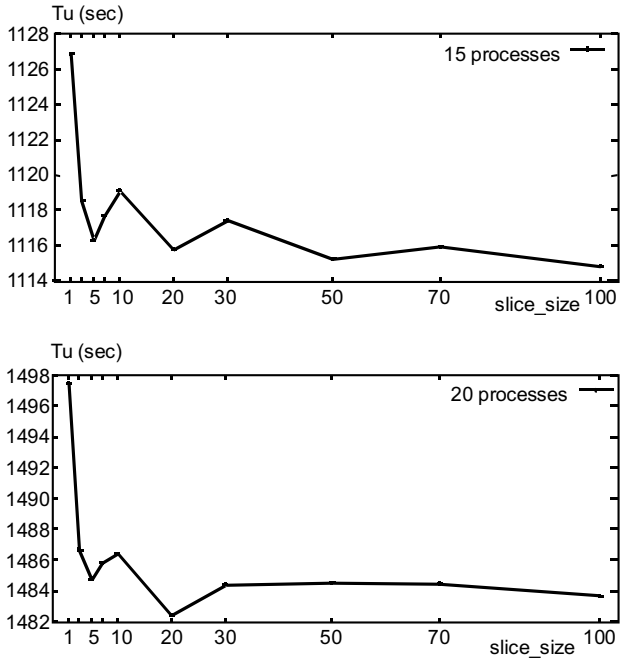


**Fig. 3.** User time as a function of *slice_size*

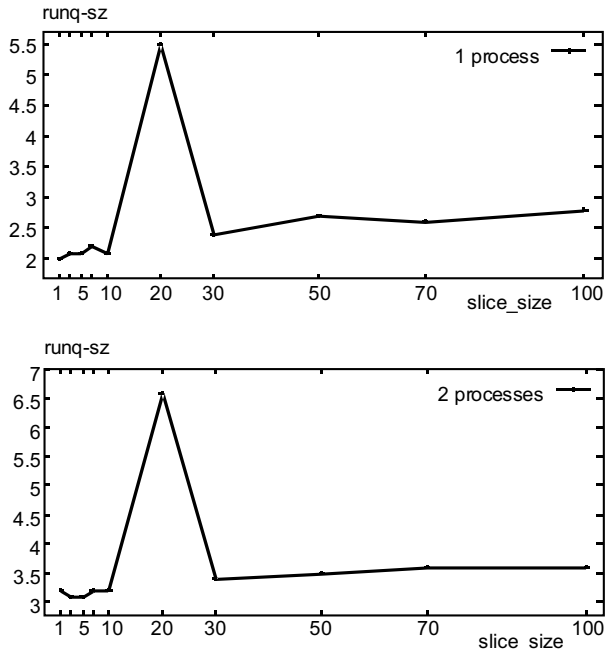


**Fig. 4.** Run queue as a function of *slice_size*

We can see that all relations are almost the same, except the case we spawn 10 processes (see Figure 5), but it is irrelevant for us because the value of *slice_size* parameter is not located within the range we are interested in. The peak located at the point of 20 were also caused by measuring inaccuracy and it is not important for us.

So, we will look through the values within the interval specified in the previous paragraph. We will try to find the minimum *runq-sz* value within this range. If we compare all relations separately for each one of them, we will find the minimum *runq-sz* values for the *slice_size* parameter at point 3.

### 4.2 Disk Management—Parameter *nbuf*

The **nbuf** [6] parameter specifies the number of buffer headers in the file system buffer cache. The actual memory associated with each buffer header is dynamically allocated as needed and can be of varying size, currently 1 to 128 blocks (512B to 64KB). The system uses the file system buffer cache to optimize file system I/O requests. The buffer memory caches the blocks from the disk, and the blocks that are used frequently stay in the cache. This helps to avoid excess disk activity. Buffers are used only as transaction headers. When the input or output operation has finished, the buffer is detached from the memory it mapped and the buffer header becomes available for other uses.

If *nbuf* is set to the default value 0, the system automatically configures *nbuf* for average systems. The *nbuf* parameter is defined in `/var/sysgen/mtune`.

We used, for the purposes of measuring, a program that creates files of different sizes, reads them three times, and finally deletes these files. We set up the program so that it tenaciously occupies the file system up to 90%. That is 300MB of the disk space available for testing, approximately.

We tested operating system response for:

a) four relatively large files (*ie*, approximately 75MB files).
b) hundred smaller files (*ie*, approximately 3MB files).
c) thousand relatively small files (*ie*, approximately 300KB files).

We changed the *nbuf* parameter values within the range of 75–6000. We kept watch over the response of the operating system in these values: 75, 600, 1200, 1800, 2400, 3000, 3600, 4200, 4800, 5400, and 6000.

We were interested in the following cumulative activity counters of the operating system, in the case of the *nbuf* parameter:

- *sys time* — the time that a process spent in the kernel mode.
- *%wio* — portion of a processor idle time with some process waiting for I/O.
- *%rcache* — cache hit ratio (reading), that is, (*1-bread/lread*) as a percentage. Note: *bread*—basic blocks transferred between system buffers and disk; *lread*—basic blocks transferred from system buffers to user memory.
- *%busy* — portion of time device was busy servicing a transfer request.

It enabled us to find out the following relations:

1. System time $T_s$ as a function of *nbuf* parameter, $T_s(nbuf)$ (see Figure 7. All relations (*ie*, for large, middle, and small files) are very similar, almost the same. We can find three minimum regions of the *nbuf* parameter values for all three cases. The first region is located

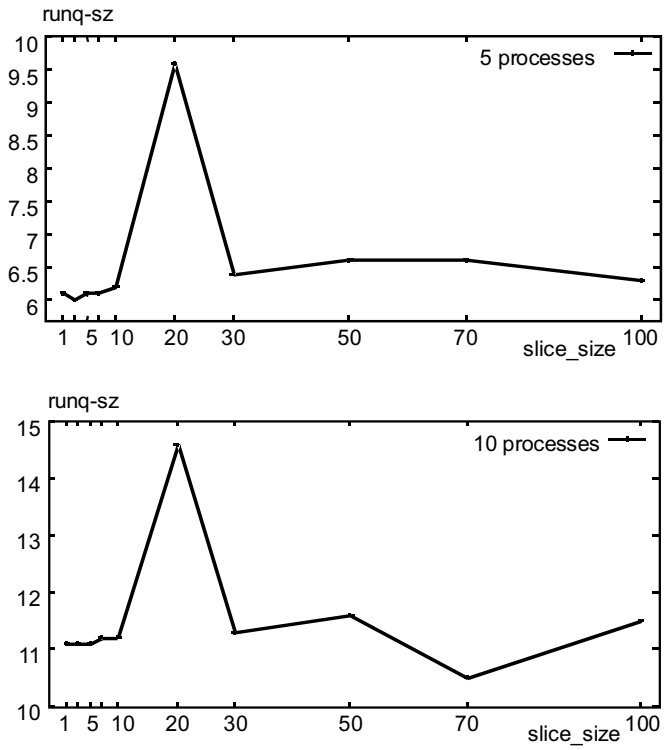


**Fig. 5.** Run queue as a function of *slice_size*

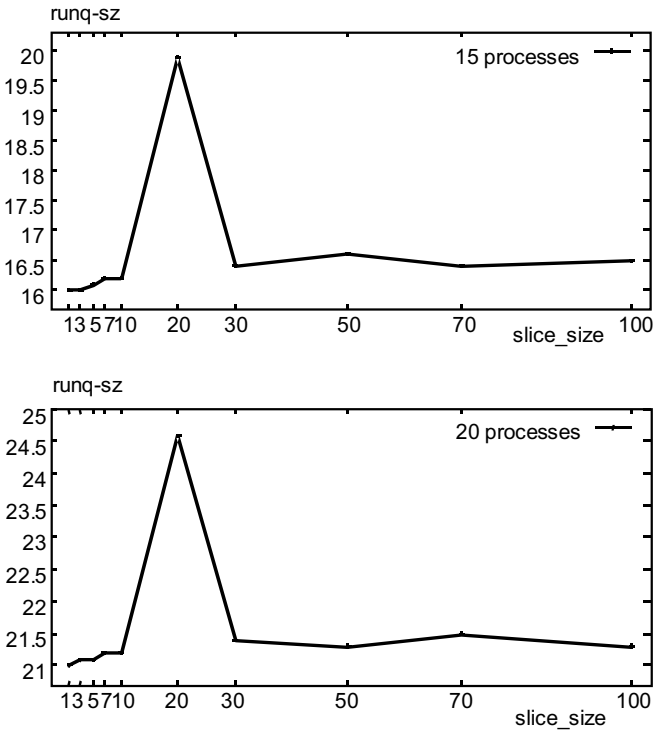


**Fig. 6.** Run queue as a function of *slice_size*

within the range of 600–1800, the second one within the range of 3000–3600, and the last one within the range of 5000–6000. We will explain later why the optimal range is 3000–3600 for our system.

It is interesting to discuss why in the *nbuf* parameter values of 2400 and 4000 are peaks, as we can see. We can suppose it is related to the particular implementation of the file system buffer cache in IRIX. We can be sure that the table of buffer headers is static and is created while booting the kernel. We can only assume that these values are critical for this kind of tasks. Unfortunately, we cannot explain it enough clearly due to a lack of availability of information about implementation of the file system buffer cache in IRIX.

2. Portion of the processor idle time with some process waiting for I/O $\%wio$ as a function of *nbuf* parameter, $\%wio(nbuf)$ (see Figure 8). As we can see, the *nbuf* parameter values within the range of 75–2000 are not favorable for the processor for all three cases we explored. Further, we see the minimum values of $\%wio$ are located in the following *nbuf* parameter values: 2400 for all three cases, 3000 for the large files, 3600 for the middle files, 4200 for the small files, and 4800 for the large and middle files. For the *nbuf* parameter values greater than 5000 we can also find good results but these values are not interesting for us (it will be explained later).

If we compare all three relations, we will find out the acceptable values of the *nbuf* parameter within the range of 2000-4800. What are optimal values from the standpoint of the operating system response for these kinds of tasks we will attempt to specify after we compare all measuring relating to the *nbuf* parameter.

3. Cache hit ratio $\%rchache$ as a function of *nbuf* parameter, this is function $\%rchache(nbuf)$ (see Figure 9). As we can deduce from all relations (*ie*, large, middle, and small files) we measured, cache hit ratio is increasing if we enlarge the number of records in the table of buffer headers. If we look at the relations in more detail, we can see a small deflection in the *nbuf* parameter value of 3600 which we can consider as unimportant.

It is important, in the case of this situation, for us to consider what the optimum is for the operating system response. The values of the cache hit ratio $\%rchache$ are within the range of 27–73. The *nbuf* parameter value of 3000 that is adequate to the cache hit ratio $\%rchache$ value about of 60 could be appropriate because it is a good enough value. But we must also take into account other results we achieved.

4. Portion of time device was busy servicing a transfer request $\%busy$ as a function of *nbuf* parameter, $\%busy(nbuf)$ (see Figure 10). As well as in the previous case, the situation corresponds to the theory. If we enlarge the number of the records in the table of buffer headers, the accesses to the device are decreasing.

Again, we have to consider what values of the *nbuf* parameter are optimal relating to I/O operations of the operating system. In this case, the deduction is almost the same as in the previous case, *ie* the *nbuf* parameter value about of 3000. Compare the Figures 9 and 10 to deduce it.

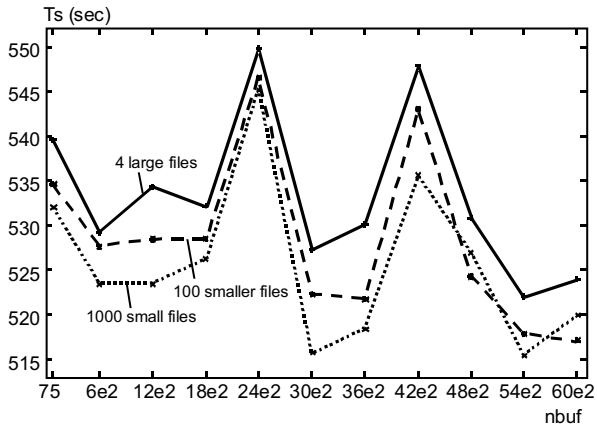


**Fig. 7.** System time as a function of *nbuf*

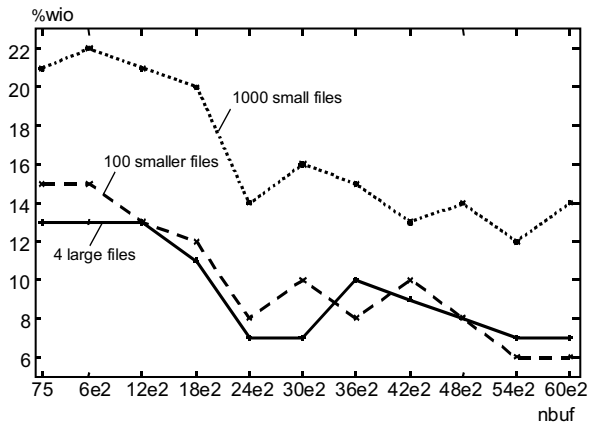


**Fig. 8.** Waiting for I/O as a function of *nbuf*

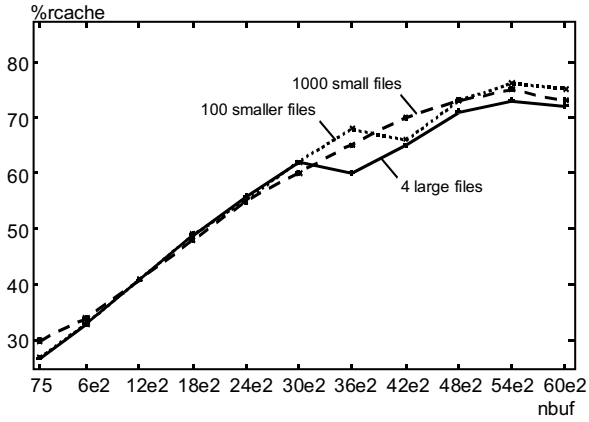


**Fig. 9.** Cache hit ratio as a function of *nbuf*

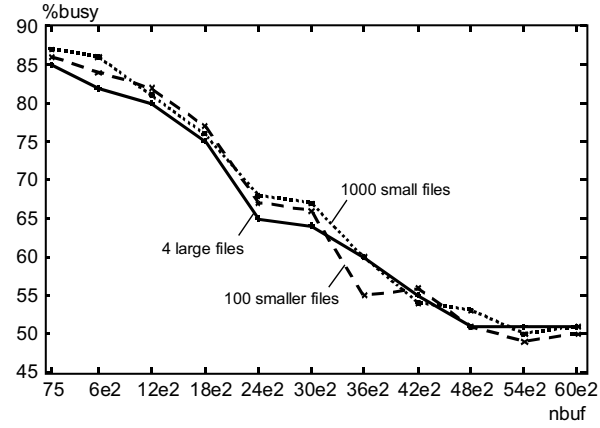


**Fig. 10.** Device busy as a function of *nbuf*

## 5 EVALUATION

### 5.1 Parameter *slice_size*

Let us see what follows from the results. First, we have to say that we monitored the system particularly for the compute-intensive (*ie*, CPU-bound) jobs. We did not pay attention to interactive response of the system.

We found that the optimal values of the *slice_size* parameter are about within the range of **3–5** (*ie*, 30–50 milliseconds). Decreasing this parameter below these values has absolutely no meaning. Of course, it is possible to increase this parameter beyond these values. It depends on what kind of jobs we want to run on the machine and what the users expect from the system.

In the case of using the system primarily for compute-intensive jobs and interactive response we can increase the *slice_size* parameter. In general, increasing this parameter gives us a greater efficiency to the compute jobs.

If the computer system is used for interactive work, we should accept the values mentioned above. But it is dependent on the particular computer system architecture. If we have a system available with more CPU units, large memory, and very efficient I/O devices, we would have more possibilities to experiment with the system.

### 5.2 Parameter *nbuf*

In order to find the optimum value of the *nbuf* parameter, we must consider all relations and results we discovered, and also take into account other aspects that play a role in the operating system tuning.

We performed measuring using the I/O-bound processes. For this kind of tasks it is typical that they spend most of their time in the kernel mode. This is why the processes perform most of their operations by the operating system servicing (*ie*, system calls). This is the reason why we paid attention to the system time $T_s$. We recovered that the three ranges (600–1800, 3000–3600, and 5000–6000) are appropriate to set up the *nbuf* parameter. There is a question which one is the right. We must compare this result with the other ones to decide what is the optimal solution.

Let us consider all measured results and compare them with each other. If we choose the first acceptable range (600–1800) from the point of view the system time $T_s$, we find the contradiction with the relations $\%rchache(nbuf)$ and $\%busy(nbuf)$ where this range is absolutely unacceptable. The same situation is in the case of $\%wio(nbuf)$ relation.

Now, we look at the second range (3000–3600) of the *nbuf* parameter we chose. Setting up this parameter within this range seems to be acceptable for all measured relations (see above).

Setting the *nbuf* parameter according to the last selected range (5000–6000) gives excellent results from the standpoint of $\%rchache(nbuf)$ and $\%busy(nbuf)$ relations. This range is also a very good choice for the relation $T_s(nbuf)$. If we look at the $\%wio(nbuf)$ relation, we find good results too.

It seems very simple to choose a right solution in the case of the *nbuf* parameter. We could say we can set it up to 5400. But we must realize that the memory requirements increase too rapidly in that case. We must consider the particular configuration of our computer system (hardware). Do we have enough physical memory to provide "so much" memory to the system buffers? We do not, in general. If enough physical memory is not available to perform some job, the operating system starts paging and swapping. Then, both memory management and I/O management is performing many operations, system workload is growing up, and the system performance is getting to be poor. Thus, we have to think about the tasks we plan to run on our computer system. For example, are they performing I/O operations only and do not require a large amount of memory? The situation can be inverse too, or we can require good performance for both kinds of tasks.

In general, we need good performance both for memory and I/O, and as we said, we often do not have enough physical memory to perform all kinds of common jobs. Then, the best setting of the *nbuf* parameter values is within the range of 3000–3600 as follows from the relations we investigated. This situation also appeared in the case of our machine configuration.

## 6 CONCLUSION

We presented basic characteristics of the UNIX operating system optimization task. We performed some tests and measurements to show how the operating system could be optimized for performance. We focused on the kernel tunable parameters *slice_size* and *nbuf*. We paid attention to the specific cumulative activity counters of the operating system according to what we wanted to monitor. We attempted to analyze our results of measuring and evaluate them to get, as long as possible, the best operating system performance.

## References

[1] BACH, M. J.: Principles of Operating System UNIX (Principy operačního systému UNIX), From American original *The Design of the UNIX Operating System* translated and supplemented by Jiří Felbáb, Softwarové Aplikace a Systémy, Prague, 1986. (in Czech)

[2] BAHYL, V.: Measuring the Performance of Operating System UNIX (Měření výkonu operačního systému UNIX), Bachelor's Thesis, CTU, Prague, Faculty of Electrical Engineering, Dept. of Computer Science and Engineering, March 1996. (in Czech)

[3] BRODSKÝ, J.—SKOČOVSKÝ, L.: Operating System UNIX and C Language (Operační systém Unix a jazyk C), SNTL — Nakladatelství technické literatury, Prague, 1989. (in Czech)

[4] DUFEK, P.: Tuning the Disk Subsystem of Operating System UNIX (Ladění diskového subsystému operačního systému UNIX), Bachelor's Thesis, CTU, Prague, Faculty of Electrical Engineering, Dept. of Computer Science and Engineering, May 1996. (in Czech)

[5] Silicon Graphics, IRIX 6.2 Reference Pages, Available in standard IRIX 6.2 configuration.

[6] Silicon Graphics, IRIX 6.2 Online Books, Available in standard IRIX 6.2 configuration.

[7] HLAVIČKA, J.: Architecture of Computers (Architektura počítačů), Lecture Notes for Undergraduate Studies, Publishing House of Czech Technical University, Prague, May 1996. (in Czech)

[8] HP-UX Performance and Tuning, Student Workbook, Hewlett-Packard Company, Mountain View, California, March 1994.

[9] LOUKIDES, M.: System Performance Tuning, O'Reilly & Associates, Sebastopol, California, November 1990.

[10] MAJIDIMEHR, A. H.: Optimizing UNIX for Performance, Prentice Hall, Englewood Cliffs, New Jersey, 1996.

[11] MOLHANEC, I.: Optimization of LINUX Operating System (Optimalizace operačního systému LINUX), Bachelor's Thesis, CTU, Prague, Faculty of Electrical Engineering, Dept. of Computer Science and Engineering, May 1996. (in Czech)

[12] PLÁŠIL, F.: Operating Systems (Operační systémy), Lecture Notes for Undergraduate Studies, Publishing House of Czech Technical University, Prague, August 1991. (in Czech)

[13] STALLINGS, W.: Operating Systems, Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[14] STEVENS, R. W.: Advanced Programming in the UNIX Environment, Addison-Wesley Publishing Company, Reading, Massachusetts, 1992.

[15] TANENBAUM, A. S.—WOODHULL, A. S.: Operating Systems: Design and Implementation, Prentice Hall, Englewood Cliffs, New Jersey, 1997.

[16] ZEMÁNEK, P.: Performance and Tuning of a Multitasking Operating System, Acta Polytechnica **38** No. 2 (1998).

Received 13 September 1999

**Milan Tichý** (Ing), born in Hradec Králové, Czech Republic in 1971, graduated in operating systems from the Faculty of Electrical Engineering, Czech Technical University in Prague, in 1999. He is a PhD student at the Department of Control Engineering. He works at the Department of Computer Systems and at the Department of Signal Processing, Institute of Information Theory and Automation.

**Petr Zemánek** (Doc, Ing, Mgr, CSc), born in Ústi nad Labem, Czech Republic in 1961. Graduated in electrical engineering from the Faculty of Electrical Engineering, Czech Technical University (CTU) in Prague in 1985 and in mathematics from Charles University in 1991. He obtained a PhD in computer science from CTU in 1990 and became associated professor at CTU in 1996. He has published more than 50 papers on various topics in operating systems, computer networks and database systems and has been a member of IEEE since 1993.