Overhead Estimation and Comparison for Multitasking Operating Systems for Personal Computers

G. Bucci and P. Nesi
Department of Systems and Informatics, Faculty of Engineering
University of Florence, Via S. Marta 3, 50139 Firenze, Italy
bucci@dsi.unifi.it, http://www.dsi.unifi.it/~bucci, tel.: +39-55-4796266, fax.: +39-55-4796363
nesi@ingfi1.ing.unifi.it, http://www.dsi.unifi.it/~nesi, tel.: +39-55-4796523, fax.: +39-55-4796363

Abstract

A method for analyzing multitasking operating system overhead and its variance is proposed. The method is supported by mathematical rationale. Results of its application to well-known operating systems (Windows 95, Windows NT, OS/2 Warp and Linux) are reported.

1 Introduction

Personal Computers are currently used for implementing real-time as well as reactive systems. Since these systems have to comply with timing constraints, the measurement (and the prediction) of the performance achievable with PCs becomes an important issue [1]. In this paper, we are concerned with the software aspect of performance. More specifically, we are interested in the evaluation of the overhead of Multitasking Operating Systems (MOS) for Personal Computers, in the perspective of their usage in real time-applications.

In selecting a MOS for a reactive/real-time application, the candidates must provide a sufficient support for implementing the system under development [3], [4]. In particular, they must allow the definition of processes, threads, semaphores, interprocess queues, scheduling policies, etc.[1].

Modeling the performance of commercial MOSs is not an easy task, since technical information concerning the implementation of the above features is seldom available: it cannot be found even in the documentation accompanying professional toolkits or red/blue books. Therefore, a black-box evaluation methodology is needed. In addition, some of the above features can be hardly modeled, because of their variability due to caching and paging [4].

[1] MOSs compliant with POSIX.4, present most of the above-mentioned features.

Some MOSs provide efficient profilers for measuring system activity (task execution time, number of context switches, etc.), but, unfortunately, profilers are intrusive and can strongly impact the outcome of the measures taken. Depending on MOS specific facilities, the influence of a profiler can be different from MOS to MOS, thus reducing the value of the comparison. On the contrary, a measure must be MOS-independent and as less intrusive as possible. In [4] several algorithms for evaluating MOSs have been introduced. However, the analysis produced consistent errors in measuring the overhead.

In the perspective of reactive/real-time applications, the most relevant performance index of an operating system is the overhead associated with task management. However, real-time applications not only require little OS overhead but also require overhead predictability, therefore the variance of the overhead is, at least, as important as the overhead itself. For the estimation of the overhead and its variance, we must define both a method for measurements and a mathematical model by which the analyst can demonstrate the relationships among the measured quantities. The mathematical model would be employed to predict bounds for the MOS behavior during the system analysis, as well as at run-time [2]. The model should also be capable of allowing a black-box evaluation of the overhead.

In this paper, we make a step towards the modeling of MOS performance. We present a method for measuring the overhead of multitasking operating systems, and describe a related benchmark program. The benchmark is portable across different MOS and can easily be employed by a generic user in order to gather overhead data. We develop an approximate, but expressive, mathematical model that provides interpretation of the measured data.

2 Estimation Approach

A *task* is a concurrent piece of code: a process or a thread. A process is an executable independent piece of code under concurrent execution with respect to the other processes and internal tasks. A process presents its independent data and resources. A thread is a procedure concurrently executed with respect to the other threads and processes. A thread is generated by a process or by another thread and shares with its creator data and/or resources. Usually, the cost for context switching is higher for processes than for threads. In the sequel we shall assume that *both processes and threads are scheduled by the MOS with an unknown preemptive policy.* An operating system implements a number of internal tasks – i.e., tasks that manage the user tasks and the MOS itself. The most important MOS internal task is the scheduler, which executes the algorithm of task preemption. Scheduler execution time and task switching time are the essential components of MOS overhead.

In the following we develop a model for MOS overhead and make a number of assumptions which will be demonstrated to be actually feasible.

MOSs manage task priority on the basis of priority. If all tasks have the same priority, the scheduler performs substantially as a round-robin algorithm. This provides a means for putting distinct MOSs in equivalent operating conditions.

Let us now consider the case in which n identical user tasks, τ, are executed on a MOS with the minimum number of active MOS internal tasks[2]. The totality of tasks is the CPU workload. The overhead is the CPU time spent by all MOS internal tasks (including the scheduler) plus the time spent in context switching. However, before proceeding, some further cautions are to be taken.

First, we must avoid the influence of the underlying hardware. Therefore, it is mandatory to obtain normalized measures or make direct comparisons on the basis of the same hardware. Talking about PCs, our attention is restricted to Intel architectures. However, in spite of the same architecture, great differences are observed from one machine to another. To exclude any hardware impact, all the experiments were performed on exactly the same machine.

Second, we must guarantee that the elapsed CPU time for each single task, T_τ, is independent of the MOS under analysis. This is obtained by running a set of identical tasks, that is, generating each of them by cloning the same piece of machine-level code. However, recent system facilities, like caches, virtual memory (pagination and/or swapping), can lead to wrong estimations since they may differently impact the different MOSs under evaluation. The effects of these facilities can be practically reduced to zero if the task code is small enough so as to show high locality, making negligible the number of cache misses, page swaps and TLB updates. This assumption can be satisfied by using a very small procedure (executed several times in each task) with a small number of local variables, such as a procedure performing some polynomial iterative calculus. In our experiments, user tasks repeatedly executed the polynomial estimation of p-Greek, (π).

Please note that the latter assumption is consistent with the target of using commercial MOSs in reactive/real-time applications. In fact, a typical real-time application requires casual or repeated execution of fast, short control algorithms, such as, for instance, a PID (proportional-integrative-derivative) control. Several benchmark tools for assessing CPU overhead and context switching have been presented in the past. Usually, they obtain estimations with errors in the order of 10-15 % – e.g., [4]. The method proposed produces more precise results and its is capable of evaluating the variability of the phenomena.

[2] The minimum number of internal tasks can vary from MOS to MOS.

3 Estimation Model

A general model can be defined by considering the execution time, $Te(n)$, as the sum of the overhead, $Ov(n)$ (as directly dependent on the execution time itself), and the elapsed CPU time for completing the n tasks, nT_τ:

$$Te(n) = Ov(n) + nT_\tau, \tag{1}$$

where

$$Ov(n) = Q(n)Te(n), \tag{2}$$

and $Q(n)$ is the overhead per second. It should be noted that definition (1) is recursive and leads to obtain:

$$Te(n) = \frac{nT_\tau}{(1 - Q(n))}.$$

Thus the overhead per second can be estimated by using:

$$Q(n) = \frac{Te(n) - nT_\tau}{Te(n)}. \tag{3}$$

T_τ **must be considered as unknown** since its estimation cannot be performed neither by counting clock of instructions nor with profiles which are intrusive. $Q(n)$ can be approximately estimated by considering $Te(1)$ instead of T_τ. This can be performed since for equation (3) we have $Te(1) = \frac{T_\tau}{(1-Q(1))}$ and $Q(1) << 1$. On this basis, an approximated version of the overhead per second from equation (3) is obtained:

$$Q'(n) = \frac{Te(n) - nTe(1)}{Te(n)}. \tag{4}$$

Then the approximated version of the overhead results to be:

$$O'v(n) = Q'(n)Te(n) = Te(n) - nTe(1), \quad (5)$$

where the second part of this formula has been derived considering equation (1) and $T_\tau = Te(1)$.
In this case, the error on the overhead per second performed by estimating $Q(n)$ with $Q'(n)$ is:

$$E_Q(n) = Q'(n) - Q(n),$$

$$E_Q(n) = -n\frac{Q(1)}{Te(n)}Te(1) = -Q(1)(1 - Q'(n)), \quad (6)$$

where the last part has been obtained by using equation (5). Since all measures are positive $E_Q(n)$ results to be negative. This means that the overhead per second estimated by using equation (4) is an underestimation of the real overhead per second: $Q(n) = Q'(n) + \mid E_Q(n) \mid$ and thus also equation (5) for the estimation produce an underestimation of the overhead $Ov(n)$. The error decrease with the increasing of the number of tasks since the $Q(n)$ increase with the number of tasks. This dependence obviously depends on the scheduling algorithm that may have a linear or higher complexity.

On this basis, it can be demonstrated that *if for two distinct MOSs, m and x, the trends of the execution time follow the relationships:*

$$Te_x(i) > Te_m(i), \qquad \frac{Te_x(i)}{Te_x(1)} > \frac{Te_m(i)}{Te_m(1)}, \quad (7)$$

for $\{i : 2..n\}$, *then:*

$$Q_x(i) > Q_m(i), \qquad Ov_x(i) > Ov_m(i),$$
$$Q'_x(i) > Q'_m(i), \qquad O'v_x(i) > O'v_m(i),$$

and $\mid E_{Q_m}(i) \mid > \mid E_{Q_x}(i) \mid$.

The above relationships guarantee that the properties that hold for approximate measurements hold as well for non-approximate quantities. The last relationship means that a bigger error per second is present by measuring the lowest overhead per second.

4 Algorithms for Measuring the Execution Time

According to the above discussion, a comparison among different MOSs can be performed by measuring the execution time, $Te(n)$, as a function of the number of tasks. To this end, we need to enforce the following two assumptions:

1st assumption: each single task τ needs to be executed by using the same CPU time T_τ in every MOS.

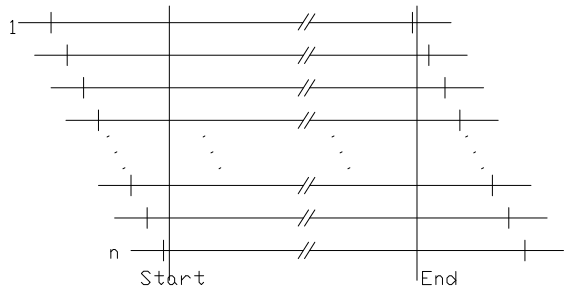

Figure 1. The Estimation Section, ES.

2nd assumption: all tasks start and finish at the same time instant.

The 1st assumption can be easily satisfied: (a) by coding the program of tasks directly in Assembly (without enabling any optimization to produce the executable program), (b) if all MOSs have been assessed on the same physical hardware.

The 2nd assumption is the more difficult to be satisfied. Usually, once the tasks have been created, they are immediately executed; this tends to produce significant differences of phases (i.e., skews) among the tasks in the workload. On the contrary, the execution time should be measured in a time interval in which the load conditions of the system are constant (i.e., with the tasks already in execution). This concept is best explained by Fig.1: the Estimation Section, ES, is comprised between the start and end instants of times. Start corresponds to the activation of the last task, while End corresponds to the termination of the first task. Between Start and End, all tasks are in execution, leading to a constant workload. In ES each task τ executes repeatedly a small piece of code. By measuring the number of times the piece of code is executed within ES, the exact CPU time consumed by each task can be computed. The executing time required by the piece of code represents the precision of the method: the shorter this time the more precise the measure.

In order to obtain reliable measures, ES must be chosen several orders greater that the MOS time slice, T_{tic} even when only one task is executed – i.e., $T\tau >> T_{tic}$.

5 Experimental Results

All the experiments have been performed on the same hardware for all MOSs, a Pentium 100 MHz, with a 256 Kbytes of cache and 32 Mbytes of main memory. The assessed MOSs are OS/2 Warp, MS Windows 95, MS Windows NT ver. 3.51, and Linux 1.2. The executable code has been produced by using Microsoft Visual C++ 2.0 for Windows 95 and Windows NT, IBM C Set ++ 2.1 for OS/2, and GNU GCC 2.6.3 for Linux. All these operating systems work in Intel protected mode.

The experiments consisted in the execution of n identical tasks τ. Each task τ performs repeated execution of the polynomial computation of p-Greek, (π), a short code which satisfies the principle of locality for cache and paging systems. In fact, no swaps from and to the disk were registered during the experiments[3].

In order to compare different MOSs, the comparison requires the same number. M, of computation of p-Greek, (π). This was done by measuring the actual number of executions performed by each task, Np, and normalizing this this number to M. The number of p-Greek estimations, M, has been fixed to 52000 corresponding to about 20 sec of execution for a single task.

Results of our experiments are reported in Fig.2 and Fig.3. In Fig.2, plots labeled with P are relative to processes, while plots labeled with T are relative to threads. Fig.3 refers only to processes.

5.1 Processes

In Fig.2 (up) the trend of the overhead as a function of n is reported. From this figure, it can be observed that the behavior is quite linear for all MOSs and, among them, Linux and OS/2 Warp are better ranked. In Fig.2 (down) the trend of the corrected overhead per second, $\bar{Q}'c(n)$, as a function of the number of tasks is shown. This figure highlights a different behavior for Windows 95 with respect to the other MOSs which have a very low increment in the overhead per second with the increasing of the number of tasks. On the contrary, this behavior is much more evident for Windows 95. This mainly depends on the scheduler of Windows 95

In order to evaluate predictability of MOS overhead, a variance analysis has been preformed (see Fig.3). This analysis is very useful to understand the behavior of MOSs with respect to the repeatability of the scheduling of the planned workload. On the basis of these results, it is evident that a sensitive difference in behavior among MOSs has been registered. Also in this case OS/2, Windows NT and Linux result to be better ranked with respect to Windows 95. For OS/2 the cost of the overhead can be quite predictable – e.g., maximizing its cost to a reasonable value over its median value on the basis of the variance. From this point of view, OS/2 Warp is the best among the MOSs compared, while Windows 95 is the less.

5.2 Threads

Linux has not been tested for threads since no official version of a support for preemptive threads was found. As regards the execution time, the OS/2 resulted to be the fastest while the lowest overhead has

[3]In addition, no mouse was moved, no communication with other machines took place (actually, net support was not installed), etc.

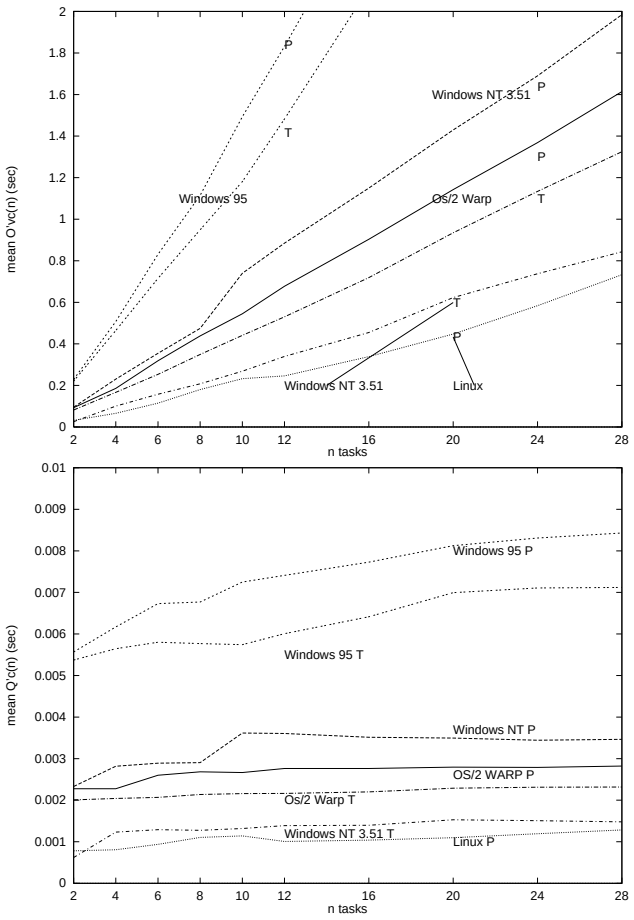

Figure 2. Overhead $\bar{O}'vc(n)$ and the overhead per second, $\bar{Q}'c(n)$, as a function of the number of Tasks (Processes/Treads) for OS/2 Warp, Windows NT, Windows 95, and Linux.

been achieved by Windows NT. From Fig. 2 it can be observed that the behavior is linear for all MOSs and, among them, OS/2 Warp and Windows NT are better ranked. For the overhead per second a different behavior between Windows 95 and the other MOSs. In fact, for these a very low increment in the overhead with the increase in the number of tasks has been obtained, while this behavior is much more evident for Windows 95.

Fig.2, shows that threads are lighter than processes. Windows NT is the operating system with the greatest difference of costs between processes and threads, while OS/2 presents the lowest difference. Windows NT threads are really light with respect to the other MOSs. The comparison among MOSs for processes and threads on the basis of the overhead per p-Greek estimation as a function of the number of tasks is reported in on the right. This figure confirms that Windows NT is better ranked than Windows 95 in supporting multitasking, since the first presents a lower cost for managing tasks. Similar results are obtained

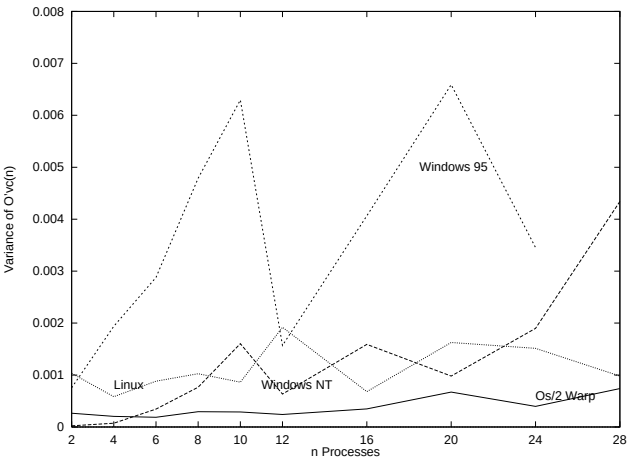

Figure 3. The variance of $O'vc(n)$ as a function of the number of processes for OS/2 Warp, Windows NT, Windows 95 and Linux.

for the overhead per second.

The real differences among MOSs have been registered about the variance of overhead time. For some MOSs problems arise when more than 6 threads or 24 processes are concurrently executed.

6 Conclusions

A technique has been proposed which allows the analysis of the system overhead as a function of the number of processes and threads. Error estimation has also been discussed. The analysis performed is not exhaustive, but provides more information than that usually available for evaluating MOSs.

Acknowledgments

The authors would like to thank M. Perfetti of Elexa and F. Butera for their help in coding the algorithms.

References

[1] G. Bucci, M. Campanai, and P. Nesi, "Tools for Specifying Real-Time Systems", *Journal of Real-Time Systems*, Vol. 8, March 1995.

[2] A. Burns, K. Tindell, and A. Wellings, "Effective Analysis for Engineering Real Time Fixed Priority Schedulers", *IEEE Trans. on Soft. Eng.*, Vol. 21, May 1995.

[3] J. B. Chen, Y. Endo, K. Chan, D. Mazieres, A. Dias, M. Seltzer, and M. D. Smith, "The Measured Performances of Personal Computer Operating Systems", *ACM Trans. on Comp. Sys.*, Vol. 14, Feb. 1996.

[4] L. McVoy, "lmbench: Portable Tools for Performance Analysis", in *Proc. of 1994 USENIX Tech. Conf.*, USA, Jan. 1994.

List of the Top and Most Popular Operating Systems with Features and Comparison. Pick the Best OS for your business or personal use from this list.Â It helps to support basic functions like scheduling tasks, and controlling peripherals. Which OS Is Best For Personal Use? When it comes to home use, traditional Windows and MAC OS are great options. At home, you donâ€™t need powerful OS especially for simple tasks like writing or browsing the web. For gaming, the Windows operating system is well optimized than that of MAC. Which Is The Fastest OS? While discussing the fastest OS, there is no argument that Linux based OS is the lightest and fastest OS in the market right now. Diagram of multitasking in operating system. As shown in the diagram above, three tasks are running on the computer. CPU gives 10 nanoseconds to each task.Â Time-sharing is the main concept and benefit of MOS. All tasks are given a suitable amount of time and no waiting time occurs for the CPU. Handle multiple users: Multiple users running multiple programs can be best handled by MOS. Embedded Systems Questions and Answers â€“ Multitasking Operating Systems. Â« Prev. Next Â». This set of Embedded Systems Multiple Choice Questions & Answers (MCQs) focuses on â€œMultitasking Operating Systemsâ€. 1. Which of the following works by dividing the processorâ€™s time? a) single task operating system b) multitask operating system c) kernel d) applications View Answer. Answer: b Explanation: The multitasking operating system works by dividing the processorâ€™s time into different discrete time slots, that is, each application requires a defined number of time slots to complete its execution When a computer has one processor, sharing time is logical and designing an OS to do so is also a logical way to go around providing multitasking capabilities.Â But for this transferring L1 cache data is an overhead which has to be taken into account as each processor will have their own cache. Some estimation algorithm can find out the time taken to do all these with the time taken to wait in same processor queue. whichever is less scheduler can choose that one.Â Most of the modern operating systems which are optimized for multi-processor architectures do this. However you can also exploit all the power of multi-core architecture by developing application that can use all the processors at the same time which is called as parallel applications. Hope it helps