A Concurrent Programming Environment for the i486

Antônio A. Fröhlich, Hélder Savietto, Luciano Piccoli & Rafael B. Ávila

Universidade Federal de Santa Catarina
Departamento de Informática e de Estatística
88.049.970 - Florianópolis - SC - BRASIL
Tel.: +55 48 231-9543 Fax: +55 48 231-9770
E-mail: guto@inf.ufsc.br

ABSTRACT

This paper describes a concurrent programming environment for the Intel's 486 processor family. The environment uses the processor's advanced features, like memory management, multitasking and protection, to supply the application level with a compact and secure execution environment.

Regarding process management, the environment supports multitasking, multithreading and dynamic priority scheduling. The memory management strategy is based on paging, which is used to map the available physical memory into logical segments for the applications. Cooperation among processes can then be achieved through shared memory and semaphores.

1. INTRODUCTION

Since IBM has chosen Intel's 8086 as the main processor for its personal computer line, this family of microprocessors has been widely used. Nowadays, most personal computers are equipped with such processor. The 80x86 microprocessor family has evolved to include several new resources, such as memory management unit, multitasking support and protection mechanisms. However, few operating systems make use of this advanced resources.

This paper describes a concurrent programming environment for PCs based on the 486 family of microprocessor: i386, i486, Pentium and Pentium Pro. The main advanced resources available in the processor are used to provide a secure, efficient and compact programming environment for applications. The proposed programming environment is extremely versatile, what makes possible for it to support the development of an operating system or to directly support applications. Besides, its reduced size code allows it to be stored in ROM, thus serving as support for dedicated systems.

This paper is organized as follow: at first, the i486 microprocessor and the environment structures are presented; next, process, memory and I/O management strategies are described; at last, the perspectives for the environment as well as authors' personal conclusions are presented.

2. THE I486 MICROPROCESSOR

Intel's i486 [7][8] is a versatile CISC microprocessor that can operate in three modes: real, virtual and protected. When operating in real mode, the processor behaves as an ordinary 8086, except by the speed. In this mode, memory is organized in fixed-size segments of 64 Kbytes each, there are no protection mechanisms, neither multitask support.

The virtual mode is an intermediate level between real and protected modes. When in this mode, protection and multitasking resources are available, but the instruction set and memory address translation still the same as in the 8086. This mode can be used to support several virtual machines, each one equivalent to a real 8086.

Protected mode is the one that makes available all processor's resources. This operating mode enables protection, multitasking and memory management. For memory management, two schemes are available: segmentation and segmentation + paging (it is really segmentation plus paging, not paged segmentation). In this mode, memory segments have variable sizes in bytes or in pages, while pages are 4 Kbytes, fixed-size elements. Regarding protection, this mode makes available four privilege levels that can be used to enforce memory and CPU access restrictions. It is also available some support for multitasking, with automatic context switch.

3. THE ENVIRONMENT STRUCTURE

The proposed environment is comprised of four modules: process management, memory management, synchronization and I/O support. Over these modules lies an interface layer that presents internal objects to application, enforcing protection. Environment internal objects are: logical memory segments, tasks, threads, semaphores, interruption handlers and I/O ports.

There are to basic way the environment can be used: to support an operating system or to support applications. Once the environment supplies only basic abstractions, it is possible to develop a complete operating system over it without any restriction. By the other hand, the environment is complete enough to support some specific applications. Figure 1 illustrates the environment structure.

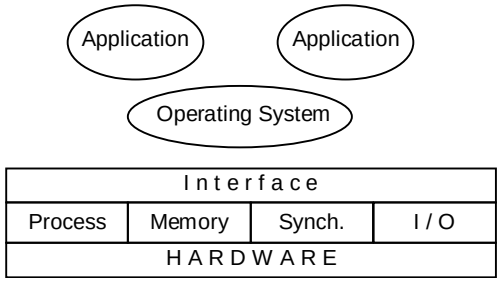

Figure 1: The environment structure.

4. PROCESS MANAGEMENT

The process manager developed for this programming environment had efficiency and flexibility as its main goals. In order to be efficient, it supports multitasking, multithreading and dynamic priority scheduling. In order to be flexible, it abdicates from most abstractions usually found in conventional systems, such as process hierarchy, ownership and grouping. These characteristics will be depicted next.

Process

Aiming for the most effective use of architecture's resources, the environment supports processes as combinations of tasks and threads [1][10], where tasks are passive entities, comprised of protected memory segments for code and global data; and threads are active entities that eventually execute some task's code. Each thread has its own context and stack, thus, a single task may present several concurrent threads. Besides supporting multithread, the environment also supports the coexistence of multiples tasks. Figure 2 presents the environment process model.

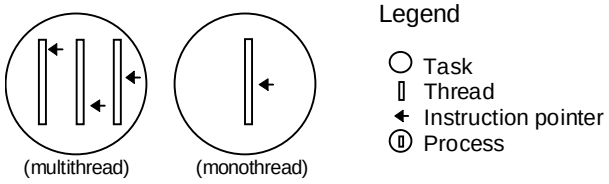

Figure 2: The environment process model.

Scheduling

The environment schedules threads independently of which task they belong to. That is, threads of a single task concur for CPU time in equality to other tasks' threads.

The scheduling policy adopted by the environment is, theoretically, dynamic priority. Nevertheless, there are some ways to influence scheduling, even from outside the environment. When a thread is created, its creator thread can define a range of priorities for it. When it is then scheduled, a timer is set to restrict its execution time to a certain limit. Every time the thread leaves the CPU, its priority is recomputed based on the portion of the time-slice it has used, and then adjusted to the interval defined for it.

This policy is similar to that adopted in UNIX operating system [3]. Such a policy has been proved to be efficient for interactive systems, as long as it benefits I/O bound process. The main difference is the existence of a priority interval, which yields to other specific policies, once the creator thread can redefine it anytime. By doing so, the environment supports the implementation of a user level scheduler.

Process Synchronization

Aiming for cooperative processing, the environment supplies mechanism for communication and synchronization. Process communication is achieved by shared memory, which is described in the next section, while process synchronization is achieved by semaphores.

Semaphores can be used to synchronize threads of a single task or they can be shared among tasks to synchronize its threads. The environment implementation of semaphore conforms to Dijikstra definitions [2], i. e., there are only two atomic valid operations on semaphores: P and V. The atomicity of these operations is achieved through i486's XCHG instruction, that atomically manipulates two memory positions.

5. MEMORY MANAGEMENT

The memory management scheme most commonly available in multitasking systems is paging. However, as stated before, i486 microprocessor does not support pure paging and a more complex scheme is required to make the segmentation transparent to applications. Actually, only two flat (entire address space) i486 segments are defined and shared by all applications: one for code and one for data. These segments have no meaning to the environment or to applications, they are defined just to satisfy the processor.

Once this flat segmentation model is established, paging can be used to implement logical paged segments for applications. Protection is then achieved through control

bits associated to each logical segment that enable or disable writing and enable or disable user level operations. Moreover, any logical segment can be expanded or shrunk, but stack segments are auto-expandable, i. e., when a stack overflow is detected, a new page is allocated to the segment. Similarly, when a stack page becomes free, it is automatically released.

Cooperation among threads of a single task is easily achieved by the shared data segment. Threads of distinct tasks can cooperate by sharing some of its logical segments. Threads can also use semaphores to avoid undesired interference.

Shared Memory

Shared memory is managed in the environment by mapping the same logical segment on different address spaces. In such a way, it is possible to share either a large segment or a single page. Besides, each process can see the shared segment in a distinct area of its address space, as memory segments are not tied to a fixed address.

Each process can share as many segments as desired. Threads of a same task implicitly share all of the task's segments, since segment allocation is handled on a per task basis.

6. I/O SUPPORT

The proposed environment does not support any I/O device, however it allows a process to request interruption handlers and I/O ports to be mapped into its address space. In such a fashion, I/O management is completely handled outside the environment by ordinary applications. By doing so, the environment remains compact and the operating system is free to manipulate I/O devices as they wish.

7. PROTECTION

In order to protect internal objects, the environment uses a capability scheme. A capability is comprised of four elements: the object class identification, the object identification inside its class, the object permissions and a random number. This capability concept was first proposed by AMOEBA [11], and is adequate to the proposed environment.

Each internal object has an associated capability, determined at creation time. Whenever a new object is created, its capability is stored inside the environment and a copy is given to the creator thread. In order to gain access to an object, a thread must present a valid capability, which is then compared to the one previously stored.

8. FURTHER IMPLEMENTATIONS

The proposed environment has been developed with extra care about portability. It has been written almost completely in C and the i486 dependent code has been isolated from the rest of the code, what makes possible for environment to migrate to other platforms, perhaps POWERPC or SPARC.

At present, a version to be used in automation as support for dedicated hardware it is being developed. This integral functionality version is expected to fit in a 64 Kbytes ROM. Besides this project, the research group is working in the consolidation of communication mechanism to be incorporated into the environment.

9. CONCLUSIONS

This paper has described a concurrent programming environment for the i486. This environment supports a minimal set of abstractions that can be used as basis for a complete operating system development or can be used directly to support applications. A first prototype is now fully operational and is being used in academia for teaching and research.

It is hard to evaluate the environment performance, because the absence of a similar system to compare. The proposed environment can not be compared to any standard operating system, because their goals and complexity differ a lot. Even micro-kernels such as Mach includes support to virtual memory and several device drives. Nevertheless, the scheduling policy has proved to impose little overhead to process execution and the shared memory scheme has also proved to be very efficient.

REFERENCES

[1] ACCETTA, M. et alli, *Mach: a New Kernel Foundation for UNIX Development*, In: Proceedings of the Summer 1986 USENIX Conference, July 1986.

[2] ANDREWS, G., *Concurrent Programming: Principles and Practice*, Redwood City: Benjamin/Cummings, 1991.

[3] BACH, M., *The Design of the UNIX Operating System*, Englewood Cliffs: Prentice-Hall, 1987.

[4] CORSO, T., *Ambiente para Programação Paralela em Multicomputador*, Florianópolis: UFSC/CTC/INE, 1993 (Relatório Técnico).

[5] ENGLER, D., KAASHOEK, M. & O'TOOLE, J., *The Operating System Kernel as a Secure Programmable Machine*, In: Proceedings of the Sixth SIGOPS European Workshop, September 1994.

[6] INTEL CO., *80386 System Software Writer's Guide*, Santa Clara: Intel Corporation, 1987.

[7] INTEL CO., *80486 Programmers Reference Manual*, Santa Clara: Intel Corporation, 1990.

[8] LEFFLER, S. at alli, The Desing and Implementation of the 4.3 BSD Unix Operating System, Reading: Addison-Wesley, 1989.

[9] STEIN, B., Projeto do Núcleo de um Sistema Operacional Distribuído, Porto Alegre: UFRGS, 1992 (Dissertação de Mestrado).

[10] TANENBAUM, A., *The Amoeba Distributed Operating System*, Amsterdam: Vrije Universiteit, 1992 (Relatório Técnico).

[11] TANENBAUM, A., *Using Sparse Capabilities in a Distributed Operating System*, Amsterdam: Vrije Universiteit, 1992 (Relatório Técnico).