UDC 004.45, DOI:10.2298/CSIS070521022R

COLIBROS: Educational Operating System

Žarko Živanov[1], Predrag Rakić[1], and Miroslav Hajduković[1]

[1]Faculty of Technical Sciences, Trg D. Obradovića 6,
21000 Novi Sad, Serbia
{zzarko, pec, hajduk}@uns.ac.rs

Abstract. This paper gives an overview of educational operating system called COLIBROS. It is small, object oriented, library operating system, based on micro-kernel concepts, supporting high level concurrency and synchronization primitives. In fact, COLIBROS is simplified operating system kernel accompanied with hardware emulation layer that emulates keyboard, monitor, disk and interrupt mechanism. A concurrent COLIBROS program behaves like stand alone program executing in emulated environment, in our case as plain GNU/Linux process. Encapsulating all critical concepts in host operating system user space makes COLIBROS development and debugging easier and more user friendly.

Keywords: Operating System, Programming Library, Education.

1. Introduction

In order to teach basic operating system undergraduate course, specially designed educational operating system is useful. This operating system should be simple enough that it can be presented with all details to students in one-semester course. Also, it should give students insight in all basic operating system concepts like multitasking, synchronization, memory management, interrupt handling, device controlling and preemption. So, we developed COLIBROS (COncurrent LIBRary Operating System) [15], educational operating system with these properties.

In the rest of this paper COLIBROS project is discussed in more details. Rationale for COLIBROS development is presented in chapter two, followed by project history in chapter three. COLIBROS implementation details such as modules, core implementation and hardware emulation are explained in chapter four. Overview of COLIBROS interface is given in chapter five. This chapter focuses on multithreading, thread synchronization, atomic variables and device controlling mechanisms provided by COLIBROS. Our conclusions and further development directions are presented in chapter six, followed by references in chapter seven.

2. Rationale for COLIBROS Development

In introductory operating system course concurrency is the key new concept that distinguishes operating system from other "ordinary" programs. Without mastering concurrency concepts it is really difficult for students to comprehend operating system behavior and functionality. Consequently, we decided to put concurrency in the center of students attention during operating system course. Examples of concurrent problems are used to give students insight in operating system internals. Though, concurrency is placed in the center of student's attention, we think that well known educational concurrency tools, like BACI [3, 4] or Hartley's java library [17], are not best shaped for operating system course because they are not adequate to present other operating system aspects.

To support our approach, we developed COLIBROS. It is designed to support execution of student programs in emulated environment on GNU/Linux platform. Students, by solving all sort of concurrent problems, in fact write different parts of operating system. We are convinced that this offers invaluable experience that prepares student minds to accept "real" problems. On that basis, it is easy to complete the whole operating system picture and help students navigate in huge number of its details.

COLIBROS offers small and simple kernel suitable for students' projects aimed to change and improve its functionality. It is similar to the well-known operating system courses during which students improve minimal kernel through assignments [2, 12, 18, 13].

Our approach differs from other well-known operating systems developed as educational tools, which are designed as fully functional UNIX-like operating systems like MINIX [24] or XINU [7]. Using our approach we directs student attention only to fundamental concepts, supported by fairly small implementation (COLIBROS core consists of 1500 lines).

There is resemblance between COLIBROS and exokernel [10, 11] library operating system. Difference is that COLIBROS is educationally oriented and it stands above hardware emulation layer (virtual machine) instead of exokernel. Without hardware emulation, standalone COLIBROS program can be executed on bare hardware. In that constellation, COLIBROS core represents simplified operating system kernel executing in the same address space with application.

Since COLIBROS is minimal operating system kernel, it offers realistic insight into crucial aspects of operating system behavior. Modifying COLIBROS core students can change system behavior influencing process, memory management or device handling.

COLIBROS exports object oriented, high level programming concurrency abstractions interface. The same abstractions are used to build operating system kernel modules. Distinguishing feature of COLIBROS is that it can be used in two ways, for teaching concurrency in high-level programs and for teaching operating system kernel internals. COLIBROS simplified operating system kernel uses threads and their synchronization to provide some typical kernel functionalities.

3. COLIBROS Project History

COLIBROS history begins with conCert (CONcurrent C for Embedded RealTime) [14]. The conCert project started in early 1990-es with intention to develop small, educational tool for operating system course at Computer and Control Department of Faculty of Technical Science at University of Novi Sad. Its goal was to support concurrency in simple multiprocessing environment. The conCert was designed as monolithic but well structured (layered), single space, library operating system. It was developed on DOS platform, using Borland Turbo C compiler, like XINU [7] or MPX-PC [20]. Putting everything in one address space, without any hardware protection between threads and kernel, made it simple and fast. During 1990-es conCert was ported to several Intel based platforms (80x86, 80960) and used as real time executive for control industrial applications.

Couple of years later, conCert was renamed to COLIBRY, redesigned to become object oriented and migrated to C++ language. Since then, COLIBRY uses objects to represent system elements (*thread*, *memory*, *ready_list*...). Concurrency (thread definition and creation) and synchronization (cooperation between threads and cooperation between threads and interrupt handlers) are implemented using C++ classes, similar to Choices [5, 6], instead of introducing new language primitives.

COLIBRY programs were executed on DOS without any hardware protection (similar to [22]). Students had difficulties developing and debugging programs in such hostile environment. Thus, COLIBRY project was migrated again [21] to GNU/Linux platform and GNU gcc compiler and renamed to COLIBROS.

During this migration it was necessary to introduce hardware emulation layer between COLIBROS core and Linux kernel. This emulation layer emulates only devices (terminal, disk) and mechanisms (interrupt handling) that are not directly/completely accessible to unprivileged Linux process. Our goal was to help students to understand real hardware and master its asynchronous behavior of hardware devices/mechanisms.

In the latest version of COLIBROS, students can use GNU/Linux memory protection and debugging tools for safe COLIBROS program development and at the same time they can freely access (emulated/real) hardware devices.

4. COLIBROS Implementation

COLIBROS consists of two layers: core (COLIBROS kernel) and hardware emulation layer. Every COLIBROS program is statically linked with COLIBROS (kernel core and emulation layer) and executed as plain GNU/Linux process (as shown in Fig 1). This organization allows independent COLIBROS processes to be executed at the same time on the same GNU/Linux host.

COLIBROS kernel is set of library functions/objects, which are linked with user code, similar to Engler's exokernel [10]. That's why it is referred to as Library Operating System. This organization allows different kernel implementations/configurations to be chosen at compile time.

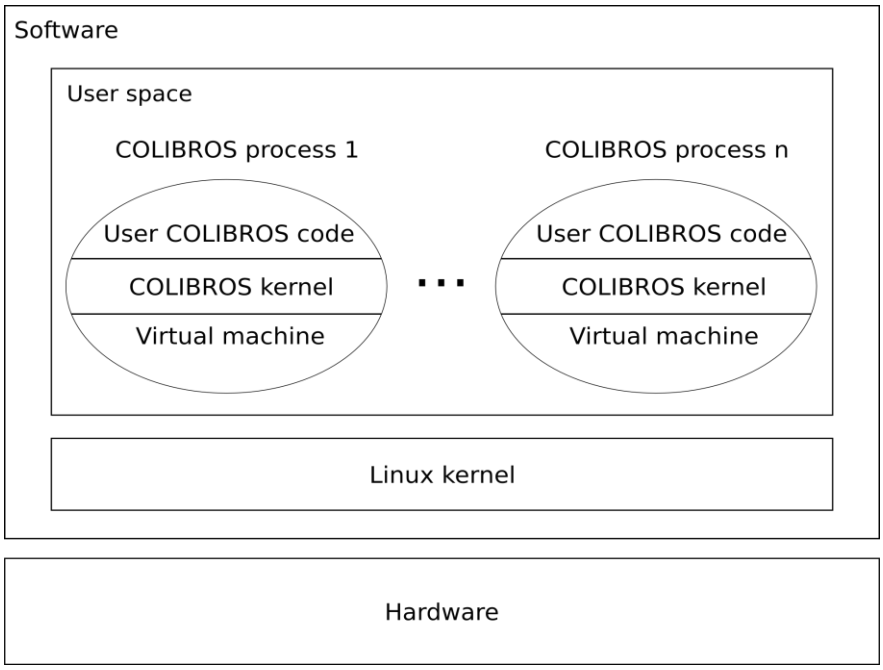

Fig 1. COLIBROS environment. Many COLIBROS processes (potentially created by different users) can coexist on the same GNU/Linux host. Every COLIBROS process executes in user space and encapsulates it's own instance of emulated hardware layer.

COLIBROS source code is divided in modules that represent logical parts of COLIBROS project.

4.1. COLIBROS Modules

COLIBROS core consists of *exec* and *in_out* modules.

The *exec* module contains platform independent COLIBROS executable source code. For every platform there should be also platform specific code. Currently there is only one platform supported: i386-pc-linux.

The *in_out* module contains character and block drivers for devices used by COLIBROS. Only terminal and disk are currently implemented.

The *exec* and *in_out* modules (without emulation layer) contain around 1500 lines of code and can be executed in couple hundred KBs of RAM.

Besides these two modules COLIBROS also contains: *tests*, *measures* and *programs* modules.

The *tests* module contains carefully designed automatic and manual test programs used during development of COLIBROS.

The *measures* module contains COLIBROS programs used to measure efficiency of different COLIBROS implementations.

The *programs* module contains examples discussed through out course. Examples demonstrate basic concurrent and parallel problems like Producer/Consumer, Dining Philosophers, Readers/Writers, Disk Head Scheduling, Parallel Sorting, Matrix Multiplication, Parallel Contour Finding and so on.

At the end it should be noted that COLIBROS is accompanied with operating system course book [15] contained in *doc* module. This book contains COLIBROS reference manual and its source code with explanations.

4.2. COLIBROS Core Implementation

COLIBROS core implementation is based on several system objects. The objects *ready* and *kernel* are in charge of processor and numerical co-processor management, and support scheduling and synchronization. These objects support context switching, preemptive round robin scheduling, mutual exclusion and conditional synchronization with sorting of thread waiting lists as well as dealing with asynchronous events (interrupts).

The objects *exception* and *timer* represent drivers that take care of exceptions and system time.

The object *memory* is due to memory management. It supports contiguous first fit memory allocation as well as C++ new and delete operators.

The objects *delta* and *wake_up_daemon* deal with thread sleeping.

Character and block input and output are supported by the objects *display_driver*, *keyboard_driver* and *disk_driver*.

These system objects constitute simplified operating system kernel suitable for changing and extending. So, during the operating system course different improvements (like advanced scheduling mechanisms, memory management techniques (e.g. virtual memory) or file system) can be added to basic COLIBROS functionality.

Other, higher level, services can be built as user threads (similar to micro-kernel architecture [24]) without changes in COLIBROS core.

4.3. COLIBROS Hardware Emulation Layer

The hardware emulation layer is added to COLIBROS during migration to GNU/Linux platform. Using it COLIBROS program is executed as plain GNU/Linux process that freely accesses only emulated hardware and therefore there is no need for real hardware access privileges.

The emulation introduces system objects that represent keyboard, display and disk controller as well as terminal. Another three system objects represent emulated interrupt table and deal with Linux signals and Linux timer.

Emulated keyboard and display controllers are software objects that together represent terminal (serial device) in raw mode. This emulation is implemented using GNU/Linux terminal primitives.

Emulated disk controller is software object that represents hardware magnetic disk device. This controller emulates small capacity, block device which stores it's data in COLIBROS process's memory. It, also, emulates some physical characteristics of magnetic disk like seek time and rotational delay.

Emulation of interrupt mechanisms requires emulation of interrupt notification mechanism, interrupt table and interrupt flag. GNU/Linux signals are used to emulate interrupt notification. Signals are designed for inter-process asynchronous communication and are well suited for emulation of interrupt notification.

Interrupt handling logic is implemented in signal handlers. Signal handlers invoke appropriate interrupt handlers (registered in emulated interrupt table) depending on emulated interrupt flag state (that enables or disables interrupt handling).

System timer emulation is based on periodic signaling provided by Linux kernel.

The emulation layer allows students to manage emulated hardware without constrains and without risk of crashing underlain operating system. At the same time, it does not hide device implementation. On the contrary, it ensures that every important implementation detail is visible.

COLIBROS emulation layer can be used on any Linux kernel based platform distribution, but install scripts are designed for and tested on Debian and some Debian-like distributions.

5. Overview of COLIBROS Interface

COLIBROS concurrency primitives are designed having in mind rich heritage of process synchronization papers [16, 25, 19, 1, 23]. COLIBROS also offers tools to synchronize threads and interrupt handlers in manner similar to ones used to synchronize threads. Besides that, COLIBROS offers flexible interface to determine order of continuation of thread activities, stopped for synchronization reasons. COLIBROS interface is intentionally designed to bare similarity to Java programming language [9] to prepare students for concurrent programming in Java.

5.1. COLIBROS Multithreading

Thread creation is conducted in three steps:

- Definition of thread class,
- Instantiation of thread object and
- Activation of thread.

Each user class that inherits COLIBROS class *Thread* represents thread.

Program 1: Thread class definition

```
class Thread {
    ...
    virtual void run(void) = 0;
    ...
    void* operator new(size_t type_size,
                       size_t stack_size = STACK_SIZE);
    void start(const int priority = DEFAULT_PRIORITY,
               const unsigned alias = 0);
    friend void destroy();
};
```

Such user class must implement member function *run()*, that contains thread body (see Program 1).

Thread objects are instantiated by C++ operator *new()*. Every thread object is therefore instance of previously defined user class.

Calling member function *start()*, inherited from class *Thread*, activates thread.

When thread activity is completed, thread objects are automatically destroyed. The friend function *destroy()* allows thread to stop its activity and terminate its existence.

The example of complete COLIBROS program shown in Program 2 illustrates definition, creation and activation of thread that prints string “Hello world”.

Program 2: Example of Complete COLIBROS Program

```
class Example : public Thread
{
  public:
    void run();
};

void
Example::run()
{
    tout << "Hello World!";
}
```

```
void
Initial::run()
{
    Thread* t;
    t = new Example();
    t -> start();
}
```

The example of complete COLIBROS program is presented as Program 2. It contains member function *run()* of COLIBROS predefined class *Initial*. *Initial* class represents initial thread, which is automatically created and started at the beginning of the program execution. The object *tout* is predefined object for the program output (similar to standard C++ object *cout*).

5.2. COLIBROS Thread Synchronization Using Exclusive Variables

Cooperation between threads is achieved by using exclusive variables. Exclusive variable is instance of an exclusive class - a user defined class that inherits COLIBROS class *Exclusive*.

Program 3: Exclusive class definition

```
class Exclusive
    class Exclusive_block {
        ...
        Exclusive_block(Exclusive* ex);
        ~Exclusive_block();
    };
    class Condition {
        ...
        void await(unsigned t = 0);
        void signal();
        bool first(unsigned* t = 0);
        bool last();
        bool next(unsigned* t = 0);
        bool attach_tag(unsigned t);
    };
};
```

Mutual exclusion is implemented in *Exclusive_block* class (see Program 3). Exclusion enter protocol is implemented in constructor and exclusion exit protocol is implemented in destructor. Creation and destruction of an object of this class begins and ends exclusive regions.

Conditional synchronization is implemented through *Condition* class. Thread might need more than one condition to be fulfilled before it can

resume its activity in exclusive region. Because of that, COLIBROS exclusive class can contain several members of *Condition* type.

Member function *await()* is intended to stop thread activity until some condition is fulfilled, and function member *signal()* is intended to continue thread activity after some condition is fulfilled. It is possible to influence order in which threads continue their execution after necessary conditions have been fulfilled. This is accomplished by linking threads (their descriptors) into waiting lists. When linked in list, each thread can be assigned value (tag), which can be used to determine order in the list. Thread lists manipulation is supported by operations: *first()*, *next()*, *last()* and *attach_tag()* of class *Condition*.

The first operation (*first()*) positions internal pointer at first thread element in list, the second one (*next()*) enables sequential moving throughout the list, the third (*last()*) positions internal pointer at last thread element in list and the fourth (*attach_tag()*) enables changing of value (that determines a thread position in the list) associated with pointed thread element.

The Semaphore Example

Semaphores are not directly supported in COLIBROS but they can be easily implemented as in Example of Semaphore Definition (Program 4).

Program 4: Example of Semaphore Definition

```
class Semaphore : public Exclusive
{
    int state;
    Condition open;
  public:
    Semaphore(int value = 1) : state(value) {};
    void wait();
    void resume();
};

void
Semaphore::wait()
{
    Exclusive_block set_up(this);
    if(state-- <= 0)
        open.await();
}

void
Semaphore::resume()
{
    Exclusive_block set_up(this);
    state++;
```

```
    open.signal();
}
```

Semaphore definition (shown as Program 4) illustrates how creation and destruction of local variable *set_up* makes exclusive region. Also, it shows how conditional synchronization is achieved by *await()* and *signal()* operations on object open of *Condition* class.

Program 5 illustrates how *Semaphore* class can be used to achieve mutual exclusion.

Program 5: Example of Critical Region Protection Using Semaphore

```
Semaphore mutex;
mutex.wait()
   ...        // mutualy exclusive region
mutex.resume()
```

The Delta List Example

As already mentioned, users can influence order in which threads continue their execution after necessary conditions have been fulfilled. For example delta list implementation (shown as Program 6) requires threads to be sorted in relative order of their expected awakening. Position of each thread in delta list depends of its sleeping time. Sleeping time of each thread is relative to its predecessor sleeping time (except the first thread with absolute sleeping time). Object delta is used by threads to access delta list.

Program 6: Example of Delta List Implementation

```
class Delta : public Exclusive
{
    Condition delta;
  public:
    void sleep(unsigned time_to_wait);
    void tick();
};

void
Delta::sleep(unsigned time_to_wait)
{
    unsigned time;
    Exclusive_block set_up(this);
    if(delta.first(&time)) {
        do {
            if(time_to_wait >= time)
                time_to_wait -= time;
            else {
```

```
                time -= time_to_wait;
                delta.attach_tag(time);
                break;
            }
        } while(delta.next(&time));
    }
    delta.await(time_to_wait);
}

void
Delta::tick()
{
    unsigned time_to_wait;
    Exclusive_block set_up(this);
    if(delta.first(&time_to_wait)) {
        if(--time_to_wait)
            delta.attach_tag(time_to_wait);
        else {
            do {
                delta.signal();
            } while( (delta .first(&time_to_wait))
                     && (time_to_wait == 0) );
        }
    }
}
```

Operation *sleep()* links calling thread at appropriate place. This place is determined during sequential advance through delta list from its beginning (*delta.first()*) towards its end (*delta.next()*). Before thread is linked in delta list (*delta.await()*), relative sleeping time of its successor is changed (*delta.attach_tag()*).

Operation *tick()* decrements first thread's sleeping time (*delta.attach_tag()*). When first thread's sleeping time reaches zero, operation *tick()* wakes up this thread (*delta.signal()*), as well as its successors with zero relative sleeping time.

5.3. COLIBROS Device Drivers

Device drivers usually consist of two parts: synchronous and asynchronous. Synchronous part is used by computer system to ask device for some kind of service. Asynchronous part is used by device to notify computer system about (usually urgent) event [8].

In traditional operating system to implement device driver one must accomplish two unrelated operations: 1) register system service (for synchronous part) and 2) register interrupt handler (for asynchronous part).

In COLIBROS these two operations are conducted through definition and instantiation of atomic variable. Atomic variable is instance of atomic class – a

user defined class that inherits COLIBROS template class *Atomic* (Program 7).

Program 7: Atomic class definition

```
template<int VECTOR_NUMBER>
class Atomic {
    ...
    class Event {
        ...
        void expect(void);
        void notify(void);
    };
    class Atomic_block {
        ...
        Atomic_block();
        ~Atomic_block();
    };
    void start_interrupt_handling();
};
```

Atomic class must implement *interrupt_handler()* member function (inherited from *Atomic_base* class). This function is registered in interrupt table under interrupt number given as template argument for class *Atomic*.

Event class encapsulates mechanism that enables threads to wait for external events (*expect()*) and to resume activity after that events arrival (*notify()*).

Consistency of atomic variables is preserved using *Atomic_block* class. Constructor of this class disables interrupt handling, while destructor is returning it in previous state.

The example of timer device driver (shown as Program 8) contains atomic class that supports registration of time (*current_ticks*) and sleeping/wakeup (*countdown*). A single thread can postpone its activity by calling member function *sleep()*, providing sleep duration time. Creation and destruction of local variable *set_up*, in member function *sleep()*, begins and ends atomic region.

Program 8: Example of COLIBROS Timer Device Driver

```
class Timer : public Atomic<TIMER>
{
    unsigned long current_ticks;
    unsigned long countdown;
    Event alarm;
  public:
    Timer(void);
    void sleep(unsigned duration);
    unsigned time_get() { return current_ticks; };
```

```
  protected:
    void interrupt_handler();
};

Timer::Timer()
{
    current_ticks = 0;
    countdown = 0;
    start_interrupt_handling();
}

void
Timer::sleep(unsigned duration)
{
    Atomic_block set_up;
    if(duration > 0) {
        countdown = duration;
        alarm.expect();
    }
}

void
Timer::interrupt_handler()
{
    current_ticks++;
    if((countdown > 0) && ((--countdown) == 0)) {
        alarm.notify();
    }
}
```

6. Conclusion and Future Work

For the last decade COLIBROS (and its predecessors) is continuously used as educational tool, providing generations of students with their first impressions of operating system details. COLIBROS joins concepts of classical and object oriented operating systems with multithreading concepts of high-level programming languages through simple and clean interface. Its implementation helps undergraduate students establish first contact with problems of concurrency, parallelism and operating systems.

Using COLIBROS in classroom has other advantages for students, like: better understanding of semantics and limitations of C++ language, introduction to Linux system calls and basic knowledge of hardware emulation (virtual machines).

In future, COLIBROS is going to be ported on bare hardware and used as small and fast object oriented, real-time (probably wireless sensor network) operating system.

References

1. Andrews, G. R., and Schneider, F. B. Concepts and notations for concurrent programming. ACM Comput. Surv. 15, 1 (1983), 3–43.
2. Atkin, B., and Sirer, E. G. Portos: an educational operating system for the post-pc environment. SIGCSE Bull. 34, 1 (2002), 116–120.
3. Ben-Ari, M. Principles of concurrent programming. Prentice-Hall, 1982.
4. Bynum, B., and Camp, T. After you, alfonse: a mutual exclusion toolkit. In SIGCSE '96: Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education (New York, NY, USA), ACM Press, pp. 170–174.
5. Campbell, R., Johnston, G., and Russo, V. Choices (class hierarchical open interface for custom embedded systems). SIGOPS Oper. Syst. Rev. 21, 3 (1987), 9–17.
6. Campbell, R. H., Islam, N., Raila, D., and Madany, P. Designing and implementing choices: an object-oriented system in c++. Commun. ACM 36, 9 (1993), 117–126.
7. Comer, D., and Fossum, T. V. Operating system design. Vol. 1: the XINU approach (PC edition). Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.
8. Corbet, J., Rubini, A., and Kroah-Hartman, G. Linux Device Drivers, 3rd edition ed. O'Reilly, 2005.
9. Eckel, B. Thinking in Java, 3rd edition ed. Prentice Hall Professional Technical Reference, 2002.
10. Engler, D. R. The exokernel operating system architecture. PhD thesis, Massachusetts Institute of Technology, 1998. Supervisor-M. Frans Kaashoek.
11. Engler, D. R., Kaashoek, M. F., and J. O'Toole, J. Exokernel: an operating system architecture for application-level resource management. In SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles (New York, NY, USA, 1995), ACM Press, pp. 251–266.
12. Gary, J. E. Using nachos is an upper division operating systems course. J. Comput. Small Coll. 18, 2 (2002), 337–345.
13. Goldweber, M., Davoli, R., and Morsiani, M. The kaya os project and the micromps hardware emulator. SIGCSE Bull. 37, 3 (2005), 49–53.
14. Hajdukovic, M. Konkurentno programiranje programskim jezikom conCert. Author's reprint, Novi Sad, 1996.
15. Hajdukovic, M. Operativni sistemi (problemi i struktura), tehnicke nauke - udzbenici ed. No. 74. Fakultet tehnickih nauka, 2004.
16. Hansen, P. B. The architecture of concurrent programs. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1977.
17. Hartley, S. J. Concurrent programming: the Java programming language. Oxford University Press, Inc., New York, NY, USA, 1998.
18. Kifer, M., and Smolka, S. Introduction to Operating System Design and Implementation: The OSP 2 Approach. Springer, 2007.
19. Lampson, B. W., and Redell, D. D. Experience with processes and monitors in mesa. Commun. ACM 23, 2 (1980), 105–117.
20. Lane, M. G., and k. Ghosal, A. Mpx-pc: an operating system project for the pc. SIGCSE Bull. 21, 1 (1989), 231–235.

21. Rakic, P. Migration of concurrent library COLIBRY from MS/DOS to GNU/Linux platform. Master's thesis, Faculty of Technical Science, Trg Dositeja obradovica 6, Novi Sad, Serbia, Feb 2006.
22. Reek, M. M. An undergraduate operating systems lab course. In SIGCSE '90: Proceedings of the twenty-first SIGCSE technical symposium on Computer science education (New York, NY, USA, 1990), ACM Press, pp. 171–175.
23. Reynolds, C. W. Signalling regions: multiprocessing in a shared memory reconsidered. Softw. Pract. Exper. 20, 4 (1990), 325–356.
24. Tanenbaum, A. S. Operating systems: design and implementation. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.
25. Wirth, N. MODULA: A Programming Language for Modular Multiprogramming, vol. 7. Eidgenössische Technische Hochschule, Institut für Informatik, 1977.

Žarko Živanov received Bachelor and Master of Science degree in Electrical and Computer Engineering from University of Novi Sad, Faculty of Technical Sciences. Hi is currently employed as assistant at University of Novi Sad, Computing and Automation Department.

Predrag Rakić received Bachelor and Master of Science degree in Electrical and Computer Engineering from University of Novi Sad, Faculty of Technical Sciences. Hi is currently employed as assistant at University of Novi Sad, Computing and Automation Department. His primary interests are: Unix--like systems, Parallel and GPU programming.

Miroslav Hajduković is currently employed as full professor at University of Novi Sad, Computing and Automation Department. He is teaching courses in Computer Architecture and Operating Systems.

Received: May 21, 2007; Accepted: October 12, 2009.