Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 2472 – 2481

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

Operating System from the Scratch: a Problem-Based Learning Approach for the Emerging Demands on OS Development

Renê S. Pinto[a], Pedro Nobile[a], Edwin Mamani[a], Lourenço P. Júnior[a], Helder J. F. Luz[a], Francisco J. Monaco[a]

[a]*ICMC – Universidade de São Paulo – Campus de São Carlos, P.O. BOX 668, São Carlos 13560-970, Brazil*

Abstract

In recent past history of computer systems industry, for decades, the hegemony of a few *de facto* standards dictated by major proprietary commercial products dominated the Operating Systems (OS) field. In such technological context, consonantly to this trend, the knowledge objective focused by academical and training courses on OS-related disciplines has often been addressed more from the stand point of essential theoretical background than of the technical skills for actuation on design and development field. Emerging paradigms, nevertheless, have been rapidly changing this scenario. Among them, the establishment of Open Source concept is boosting the growing diversity of new operating systems; concomitantly, evolution of embedded hardware architectures has make it possible to run sophisticated operating systems where only bare rudimentary, ad hoc system-software were once practical. Aligned along this perspective, this paper introduces a new platform for teaching and training programs on OS development founded on a project-based approach which guides the student throughout the process of designing and programming a sufficiently simple, but yet realistic and fully functional, OS from the scratch. The differential of the present proposal regarding related works is that, instead of either merely inspecting example-code or experimenting with simulators, the apprentice is guided across the challenge of coding an entire new instance of a didactic system specification. A comparison of the companion OS-example with existing alternatives brings out a less complex implementation structure which maps conceptual modules with implementation blocks in an intuitive correspondence and with reduced function coupling. Moreover, the learning platform comes with a courseware material consistently linked to the laboratory practices, and aimed at the systemic comprehension of the many related multidisciplinary aspects.

Keywords: Operating Systems; Computer Science Education; Problem-based learning

1. Introduction

After the emergence of the first operating systems during the early days of commercial computing era, the practical ubiquity of a few very successful products that dominated the industry for over two decades significantly restricted the activity on OS development out of the domain of major players. Such trend was then followed by a corresponding resettling of the relevance of the OS-related disciplines in academic curriculum more towards the theoretical background than towards technical skills for professional actuation on the field. This scenario, nonetheless, is rapidly changing. The dissemination of the open source development model — with the noteworthy diversification of novel industrial-standard operating system — and the evolution of the embedded hardware

*Corresponding author. Tel.: +55-016-3373-8631 ; fax: +55-016-3371-9633.
E-mail address: rene@icmc.usp.br.

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.424

platform capable of running sophisticated operating system have both contributed to a trend of increase in the demand for professionals to actuate directly on OS development. This includes the design of new operating systems, porting of legacy OSes to new hardware, device driver implementation and other specialized technical activities where the effective knowledge of both theoretical and practical aspects of operating system design and implementation is relevant, specially in view of the challenges of novel architecture paradigms and requirements of distributed, mobile, pervasive, real-time and critical applications.

Along with the conceptual approach to the main aspects important to the comprehension of OS theory, a strategy which has been increasingly employed by more advanced courses is introducing the student to laboratory practices involving programming exercises. These usually include experimenting with algorithms with the aid of OS simulator or exploratory intervention on example source-code of real operating systems. Widely known examples include program instructions based on Tanenbaum's MINIX [1] educational OS and the regarded GNU/Linux project, increasingly adopted as case study in academic text books, due to its recognized importance and its open source licence. Nevertheless, although well exploited in this context, the experience of make small changes in parts of a complex operating system, such as Minix and Linux, is still less rich than that of designing and building an entire operating system in terms of the systemic view of the whole and the relations between the many conceptual and practical aspects involved. Specifically, the connection of the theoretical problems and algorithms, on one hand, and the engineering techniques available to their technological implementation and integration, on the other hand, can appear fragmented if not properly correlated during the course of the instruction program.

This work introduces an educational platform for courses on operating system development that address this challenge by a project-based learning strategy in which the student is conduced throughout a step-by-step laboratory exercise of designing and implementing an entire, fully functional, operating system from the scratch.

Among the reasons why this is admittedly challenging is the notoriously broad scope of specific technical aspects involved, including hardware specificities, interrelation of many subsystems and low-level programming and architecture-dependent issues. The construction of the many functions of even a basic OS requires a solid multidisciplinary background, which encompasses from hardware architecture and a bit of electronics to assembly programming and practical approaches to classical algorithmic problems. Although there is a vast amount of material on the Internet with useful information for the execution of such ambitious project, those sources are mostly sparse in papers, tutorials, informal reports and abundant source-code examples. A systematic compilation of all this information into a didactic guide for laboratory practices, which encompasses all the process steps of constructing an simple, yet realistic, fully functional operating system is not available in the form herein proposed.

The experience with the learning platform reported in the present paper is grounded on the specification of a OS architecture whose structure, at the implementation level, maps a simple and coherent correspondence between conceptual topics and source code. This design is conceived so as to present a step-by-step instructional program made of learning modules sequentially connected and directly linked to the implementation modules. The implementation architecture, therefore, aims at minimizing the functional coupling between subsystems and function calls, and obeys a perspective of incremental development of the source code according to an intuitively connected sequence referring to the didactic material.

The platform includes also an abundant documentation, both external and internal, which explains each part and offers a guided tutorial to implement an entire OS according to the specification. In addition, a companion instantiation of this specification, called the TempOS operating system, for a standard Intel IA-32 (x86 32 bits) architecture, is also provided and referenced by the didactic material. This work has been carried in the scope of a research project withing the Laboratory of Distributed Systems and Concurrent Programing of SSC-ICMC USP[1], and of the National Institute of Science and Technology on Critical Embedded Systems INCT-SEC[2], where it is expected to be used as a training material for development of embedded and real-time operating systems.

2. Related Work

A review on the curricula of some regarded academic courses on Operating Systems reveals several initiatives aligned to the perspective of this work. The regarded text-book "Operating System Design and Implementation"[1]

[1] www.usp.br
[2] www.inct-sec.org

(first edition published in 1987) is a classical courseware adopted by many computer science courses. Nowadays, there are also other books which uses Linux source code as study case[2]. Other similar iniatives, described in the literature, are usually performed with the aid of educational Operating Systems, most of them based on simulation, i.e. algorithms which are evaluated in hardware architecture emulators[3]. A good example of this is Nachos[4], an educational Operating System developed at Berkeley University. The kernel runs over other OS, simulating a MIPS R2000/3000 architecture. In the described approach, students receive just a skeleton code, to which they are required to develop and add features not implemented. The OS/161[5] is another education OS, developed at Harvard and based on their experiences with Nachos, aiming to overcome some of its limitations.

Among non-emulated educational Operating Systems, i.e. which are executed directed on real hardware, there are many examples of OS for embedded architectures. Xinu[6] was developed in the 1980s at Purdue University; despite being the main theme of two text books, currently it's not being developed anymore[7]. Topsy[8] it's a micro-kernel OS developed by Swiss Federal Institute of Technology in the Computer Engineering and Networks Laboratory. According to [7], its usage seems to be restricted only to few European colleges. Topsy runs on a MIPS R3000 microprocessor, either on physical hardware or on simulators. For more complex platforms, as the conventional PC architecture, there is one of most famous Operating Systems project: Minix[9], an Unix clone written by Andrew S. Tanenbaum between 1984 and 1897. At that time, Unix was open and used by many universities in their OS's courses. Starting by version 7, AT&T changes the software license, banning the studying of Unix source code, so Minix became an important alternative. Although architecturally more modern (based on *micro-kernel* architecture), Minix are seen my many more as an academic concept than as real production OS, being a big discussion point of the well known Tanenbaum-Torvalds debate Minix vs. Linux[10] — apropos, Minix, since version 3.0 is not long and educational-purpose, according to the official FAQ[11]. In the set of case studies, the usage of Operating Systems in undergraduate courses are based on simulations, modification of existent kernels or on the implementation of very specific functions.

3. The TempOS platform

One major difficulty to conduce a course on Operating System through the full step-by-step source code programming is that the implementation of its several essential features, in a complex system, can be found very fragmented along many parts of the source code, instead of conceptually structured in cohesive modules. Issues concerning performance and resource efficiency, essential features for most OSes, may recommend architectural decisions that result in a strong interdependence of different components, so that a particular functionality is performed by complex interactions among a lot of procedures and program functions that share states. If on one hand it is possible to apply object-oriented modeling in the architectural design, the real implementation has yet to be mapped in a low level vision system, which means interacting with the hardware and its particular specificities – not always well abstractly encapsulated in a higher level view. Thus, it is often not trivial to elaborate a roadmap of classes that connects concepts to the implementations in a simple way, and which relates the addressed content to self-contained parts of source code (Figure 1).

The strategy of the proposed learning platform, which accounts for its differential regarding related works, is based on the conception and specification of an OS architecture at the implementation level, in which the component modules are designed to minimize the source code coupling, and to present a simple and direct relation with the conceptual topics highlighted at the theoretical study (Figure 2). Ideally, the study of the source code should be facilitated by a instructional program linked to concise modules with low interaction with other parts, specially if those are to be addressed in later chapters. This is also true for the implementation exercise, if the complete architecture can be designed incrementally, from the parts already added to the system, diminishing the effort to resort to source fragments throughout the entire program, not intuitively related.

The proposed platform consists of the system specifications, description the learning and training method and an implementation instance which exemplifies the covered content. Armed with this resource, the student shall be able to reimplement by himself the whole architecture, with the possibility to exercise his knowledge by doing modifications and functional extensions. It is a general requirements of the aforementioned system the simple and comprehensive architecture design, didactic and readable code, comprehensive and precise documentation.

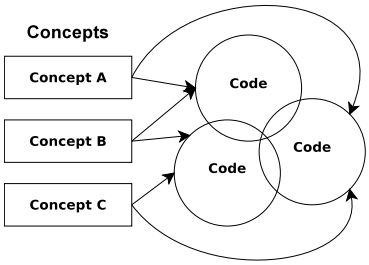

Fig. 1. High interdependence between concept and implementation.

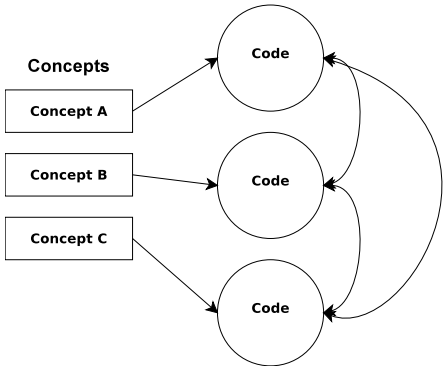

Fig. 2. Lower interdependence between concept and implementation.

3.1. Platform components

The platform, called TempOS, is composed by the following components:

- a complete architecture specification of a functional Operating System, based on Unix (monolithic kernel), designed for IA-32 Architecture (Intel x86) and composed by specific modules, including
  - system kernel, which provides a internal (for device drivers) and external (for system calls) API following POSIX standard, besides manage all basic machine recourses.
  - device drivers for general devices (like keyboards and IDE disk), and for specific devices, such as Programmable Interrupt Timer (PIT) and Programmable Interrupt Controller (PIC);
- source code of an Operating System which implementation follows the platform specification, exemplifying implementation techniques and project approaches for each system component;
- a complete guide divided into practical activities, which by the using of the platform model, leads the students through a build process of a basic (but functional) Operating System, since the first lines of code up to more complex modules, including booting process, hardware access, memory management and other high level functions. The classes sequence is designed to be used as the based courseware for undergraduate computer science and engineering courses.

3.2. Platform Architecture

The TempOS architecturecorresponds to the Unix architecture presented in Bach (1986)[12] with some adaptation. It has a monolithic kernel adherent to the POSIX standard system calls. The Virtual File System layer allows supporting several file systems and it's based on Linux's VFS layer. The architecture is composed by six main modules: system call interface, virtual file system layer (VFS), buffer cache, device drivers, hardware control and process control subsystem.

3.3. The TempOS Operating System

The OS-example accompanying TempOS platform, also named TempOS[3] was written for IA-32 (x86 32 bits) architecture, 93.33% in C language, using assembly (AT&T syntax) strictly for architecture dependent code. With a simple compilation system using conventional *shell scripts* and the *make* tool), *TempOS* can be built and tested usually flawless from any GNU/Linux system, and can be either run on an emulated environment with QEMU[4] or running on real hardware (PC). Comments represent 40.63% of the code and contains references to other learning

[3]TempOS stands for "TempOS is an educational and multi purpose Operating System", playful recursive acronym in homage to GNU system.

[4]Open source processor emulator, available at http://www.qemu.org

material (such as for Intel IA-32 manuals). The source code was structured keeping in mind the decoupling of theoretical concepts at the source code level. Figure 3 shows the organization of the source code related to each module of the architecture specification. Files are expressed by ellipses, directories by polygons and the architecture modules by rectangles. The *arch* directory contains all architecture dependent code. Other files not direct related with the modules were omitted for clarity.

Fig. 3. The organization of TempOS source code with modules relationships.

The source code is available at *http://tempos-project.org* under a GPL license and is totally modularized, enforcing an effective isolation of x86 architecture-dependent code. Among its features *TempOS* has a multiboot system that can be handled by GRUB[5], a standard C function library, dynamic kernel reallocation (linked at 3GB), flat memory addressing organization, shared IRQs, device drivers for keyboard and generic PATA controllers, POSIX system calls, Virtual File System Layer, partial support for EXT2 File System, fully support to partition tables, kernel threads, round robin scheduler and other basic resources present in production Operating Systems. After its initialization, *TempOS* loads the *init* program onto the RAM and starts to run it in user space. Through doxygen tool[6] a complete source code documentation (for functions structures, data structures specification, etc) can be generated in HTML or PDF.

TempOS OS follows the *TempOS* platform architecture specification and the companion courseware describes the process whereby the operating system is developed. The study guide is composed by ten conceptual topics linked sequentially, referencing practical "step by step" implementation exercises[7]. Following these steps, the student is guided by an iterative developing process leading to the incremental building of a POSIX operating system. Figures 4 and 5 shows *TempOS* running on QEMU emulator and one of the pages of internal kernel documentation.

[5] http://www.grub.org.

[6] Available at http://www.doxygen.org

[7] Available at [13] or `http://www.teses.usp.br/teses/disponiveis/55/55134/tde-06122012-105021/pt-br.php`. Language translation is on progress as a volunteer effort by contributors.

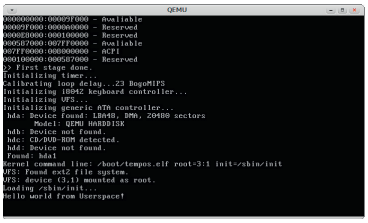

Fig. 4. TempOS running on QEMU emulator.

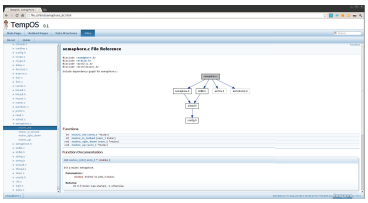

Fig. 5. TempOS kernel internal documentation.

4. *TempOS* and related approaches

Linux and Minix are two OS's being used in courses with practical approaches. However, version 2.6.30 of Linux kernel has more than 11 million of lines of code[14] and Minix is not much shorter. To bring out some notion of *TempOS*' dimension and structural complexity, a comparison can be made considering the source code of a very early version of Linux (version 0.99.15) and the lasted educational-purpose Minix (version 2.0.0). It's important to note that both systems, even in their older and smaller versions, encompass far more features than *TempOS*, as well as a lot more device drivers, a full TCP/IP stack and other resources. Naturally, their source codes are therefore incomparably larger. Having a shorter source code, in this aspect, is purposely proposed by *TempOS* as an advantage: it is easier to read and to understand, at the cost of lacking sophistication and performance improvements of the other two kernels. A question that can immediately come to mind is then why not just using a "shrinked" version of a legacy operating system? Indeed this idea was also considered and even tried. Nevertheless, the effort to strip a production kernel such as Linux from "unwanted" parts is prone to result either in a overwhelmingly impracticable effort, or in a non-functional collection pieces of code – writing a new OS came out as a more efficient approach. Furthermore, the code decoupling and the clarity vs. performance trade-off was a major concern not always addressed with the same perspective by production operating systems.

Figure 6 shows the graphic for total lines of code (including comments and blank lines) and the total lines of comments for each evaluated Operating System. The percentage indicates total amount of comments for the whole source code. Regarding the evaluated sources, Linux was the biggest kernel, with 120874 lines of code, followed by Minix, with 17421 and *TempOS*, with 6236 lines of code. Regarding the comments, *TempOS* source code presented the major portion, being 40,63% of the whole sources composed by comments, followed by Minix, with 21,12% and Linux, with 18,17%.

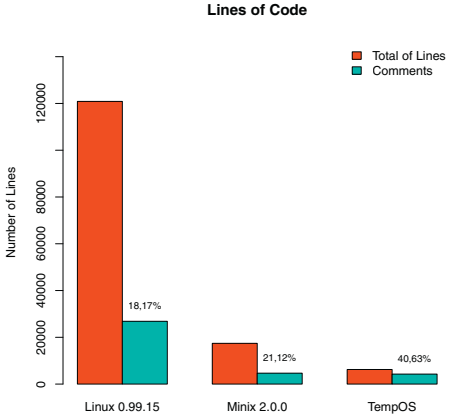

Fig. 6. Number of Lines of Code and Comments

5. Utilization example

Keeping in mind the implementation focus of the platform, the direct relation of TempOS source code and the theoretical conceptual and architecture modules is meant to facilitate the learning and understanding of the theory. For instance, when studying the OS process scheduler implementation, a student inspecting Linux might be required to go through a piece of code like that implemented by the complementary functions *schedule()* and *init_scheduler()*. *TempOS* implements those same functions, but enforcing the didactic requirements proposed by the platform. For instance, TempOS scheduler study is performed by the *schedule()* and *init_scheduler()* functions as show bellow.

```
void init_scheduler(void (*start_routine)(void*))
{
        /* Create circular linked list */
        c_llist_create(&tasks);

        /* Start scheduler time counter */
        sched_cnt = jiffies + scheduler_quantum;

        /* Architecture dependent */
        arch_init_scheduler(start_routine);
}
```

The *init_scheduler()* function initializes the rotate list, sets the time at a new task will be swithed (*jiffies* + *scheduler_quantum*) and calls the routine to execute the architectural dependent code, that transforms the passed function (*start_routine*) in the initial *kernel* thread. The *schedule()* function is called always that the time quantum is reached. The next task choice is done using *Round-Robin* algorithm, i.e., the next task of the list that is in the "READY_TO_EXECUTE" will be put to execute by using the function *switch_to()*, as in the following code, where the macro GET_TASK only returns the pointer to the task structure from the pointer of the input in the rounded list:

```
void schedule(void)
{
        task_t *c_task;
        c_llist *tmp, *head;

        if (cur_task == NULL) {
                return;
        }

        /* do schedule */
        tmp = head = cur_task;
        do {
                tmp = tmp->next;
                c_task = GET_TASK(tmp);
                if (c_task->state == TASK_RUNNING ||
                                c_task->state == TASK_READY_TO_RUN) {
                        switch_to(tmp);
                        break;
                } else {
                        continue;
                }
        } while(tmp == head);
}
```

The Linux scheduler (0.99.15), in turn, makes use of the same functions and shceduling policy. However *sched_init()* is implemented as follow:

```
void sched_init(void)
{
        int i;
        struct desc_struct * p;

        bh_base[TIMER_BH].routine = timer_bh;
        if (sizeof(struct sigaction) != 16)
                panic("Struct sigaction MUST be 16 bytes");
        set_tss_desc(gdt+FIRST_TSS_ENTRY,&init_task.tss);
        set_ldt_desc(gdt+FIRST_LDT_ENTRY,&default_ldt,1);
        set_system_gate(0x80,&system_call);
        p = gdt+2+FIRST_TSS_ENTRY;
        for(i=1 ; i<NR_TASKS ; i++) {
                task[i] = NULL;
                p->a=p->b=0;
                p++;
                p->a=p->b=0;
                p++;
        }
        /* Clear NT, so that we won't have troubles with that later on */
        __asm__("pushfl ; andl $0xffffbfff,(%esp) ; popfl");
        load_TR(0);
        load_ldt(0);
        outb_p(0x34,0x43);              /* binary, mode 2, LSB/MSB, ch 0 */
        outb_p(LATCH & 0xff , 0x40);    /* LSB */
        outb(LATCH >> 8 , 0x40);        /* MSB */
        if (request_irq(TIMER_IRQ,(void (*)(int)) do_timer)!=0)
                panic("Could not allocate timer IRQ!");
}
```

As in that version Linux has used the task switching mechanism of the x86 architecture, the function *sched_init()* starts a TSS structure in the available GDT entries. After, the flag NT was cleaned up and the TR registers (*Task Register*) ant the LDT table are loaded. Finally, the PIT registers are configured and the routine to IRQ0 treatment is registred though *register_irq()* function. As for the *schedule()* function, it not only performs the scheduling but also checks the blocked tasks waiting for a signal or alarm, and those can be "waked up" loading debugging values to certain registers. It is implemented by the following code:

```
asmlinkage void schedule(void)
{
        int c;
        struct task_struct * p;
        struct task_struct * next;
        unsigned long ticks;

/* check alarm, wake up any interruptible tasks that have got a signal */

        cli();
        ticks = itimer_ticks;
        itimer_ticks = 0;
        itimer_next = ~0;
        sti();
        need_resched = 0;
        p = &init_task;
        for (;;) {
                if ((p = p->next_task) == &init_task)
                        goto confuse_gcc1;
                if (ticks && p->it_real_value) {
                        if (p->it_real_value <= ticks) {
                                send_sig(SIGALRM, p, 1);
                                if (!p->it_real_incr) {
                                        p->it_real_value = 0;
                                        goto end_itimer;
                                }
                                do {
                                        p->it_real_value += p->it_real_incr;
                                } while (p->it_real_value <= ticks);
                        }
                        p->it_real_value -= ticks;
                        if (p->it_real_value < itimer_next)
                                itimer_next = p->it_real_value;
                }
end_itimer:
                if (p->state != TASK_INTERRUPTIBLE)
                        continue;
                if (p->signal & ~p->blocked) {
                        p->state = TASK_RUNNING;
                        continue;
                }
                if (p->timeout && p->timeout <= jiffies) {
                        p->timeout = 0;
                        p->state = TASK_RUNNING;
                }
        }
confuse_gcc1:

/* this is the scheduler proper: */
#if 0
        /* give processes that go to sleep a bit higher priority.. */
        /* This depends on the values for TASK_XXX */
        /* This gives smoother scheduling for some things, but */
        /* can be very unfair under some circumstances, so.. */
        if (TASK_UNINTERRUPTIBLE >= (unsigned) current->state &&
            current->counter < current->priority*2) {
                ++current->counter;
        }
#endif
        c = -1;
        next = p = &init_task;
        for (;;) {
                if ((p = p->next_task) == &init_task)
                        goto confuse_gcc2;
                if (p->state == TASK_RUNNING && p->counter > c)
                        c = p->counter, next = p;
        }
confuse_gcc2:
        if (!c) {
                for_each_task(p)
                        p->counter = (p->counter >> 1) + p->priority;
        }
        if(current != next)
                kstat.context_swtch++;
        switch_to(next);
        /* Now maybe reload the debug registers */
        if(current->debugreg[7]){
                loaddebug(0);
                loaddebug(1);
                loaddebug(2);
                loaddebug(3);
                loaddebug(6);
        };
}
```

In contrast to *TempOS* scheduler code, Linux routines make no strict isolation of architecture-dependent routines, and refers to functions with purposes not strictly necessary to the understanding of the scheduling mechanism itself, such as alarm and signals checking, in addition to programming specific hardware (PIC and debug

registers). *TempOS* scheduler code does not clutter the routine with functions not specific to the theoretical module, which can be explored opportunely.

6. Results and Conclusions

The development of TempOS platform started in 2008 and the first official release was available in 2012. During this period, the system was explored in a few pilot experiences in the scope of the undergraduate courses at the ICMC-USP. In 2009, when the platform was in still in its early stage of development, TempOS was first experimented in the context the discipline "Operating Systems II", for the students of the 4th year the Computer Engineering course. The proposed class program was to challenge students to develop a primitive kernel for Intel IA-32 architecture, which should run in a real hardware (PC) and execute basic routines in kernel space. The kernel should run in real mode, implementing interrupt control, basic drivers, such as PIC/PIT, keyboard and text-mode video drivers. A small shell to support basic commands, only to illustrate kernel execution, should also be implemented. Since the platform courseware guide was not complete on that time, students were allowed to resort to *TempOS* source code for architecture-dependent parts.

Originally 17 students were enrolled into the course. The very first lesson learned from trying out our approach was a simple, yet disturbing one: Operating Systems scare computer sciences/engineer students. Such observation became evident when, no less then 36% of the students simply dropped the course after the first day of class right after the didactic program announcement — and that without raising any questions or complaints. After the fact, when the event was reported to the course instructor, inquired about the reason for such a radical decision some students argued that the mere idea of designing an OS was "terrifying" enough for them to consider themselves capable of such an "overambitious endeavor", reserved to "guru hackers only" — to paraphrase some heard aexpressions. Not a pleasant finding in the context of a graduate course whose aim is to prepare future professionals for pushing technological development towards new achievements.

The 11 remaining students, however, completed the entire course and sucessfully accomplished the mission. Table 1 show the distribution of the final scores (minimal score to complete the course is 5.0).

Table 1. Students grades of Operating Systems course using TempOS platform

Grade	Number of students
5.0 - 5.9	2
6.0 - 6.9	2
7.0 - 7.9	0
8.0 - 8.9	2
9.0 - 9.9	5

After this preliminary experience, the *TempOS* has been exercised a few times, but always as extra-class activity, while the platform deployment strategy gains maturity and can be considered to officially integrate the discipline curriculum. In the last essay, carried out in 2012, subscriptions were open for a one-semester extra class course on OS development. A total of 20 participants were selected among undergraduate students from the courses of Computer Sciences and Computer Engineering. The course program covered the process of design and implementation of a functional OS for x86 32-bit platform, from the boot to user-space process execution. The program included also the implementation of a simplified *libc* port with some POSIX functions (*fork*, *exec*, *read*, *write* and a simplified *printf*.) With the library and a cross complier (gcc) the students were supposed to program a very simple shell with basic features such as file system manipulation and program execution.

Out of the 20 participants, 14 completed the course; 6 dropped after 3 months — inquired about the reason, 5 of them argued upon the need to study for other formal disciplines and 1 engaged in a trainee program. The remaining students accomplished the goals with different levels of success. All of them were able to implement an operational kernel with the required functionality — no line of code from *TempOS* example were reused. Only 6 were able to port the lib and write the shell, what corresponds to 30% of the participants. This result was considered reasonable for the purpose of the experience, bearing in mind that the course was run as a completely

optional activity, neither counting credits nor offering any certificate. The next reproduction of the experiment shall include a systematic qualitative assessment based on a methodologically prepared question sheet.

7. Final Remarks

TempOS platform is one of the outcomes of a long-lasting effort to update the didactic program of OS-related disciplines offered by the SSC-ICMC USP to undergraduate students of the courses on Computer Sciences and Computer Engineering. As a first step, it is expected to be formally integrated into the curriculum of the discipline "Operating System II" as the main laboratory practice. In the medium term, this can also be extended to the former discipline "Operating System II" under the perspective of a problem-based learning approach also for OS theory. A long term goal might be to merge those two disciplines, along with "Computer Organization", into a series of connected courses with integrated theory-practice approach. The next goal towards this direction is extending the didactic material compiled for the *TempOS* platform into a text-book for courses on Operating Systems and Computer Organization, currently in elaboration by the authors of this article. It is expected to be an option to undergraduate, graduate and professional course instructors interested on a hands-on approach to computer system disciplines.

Acknowledgements

Authors are grateful to FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo), CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico), CAPES (Coordenadoria de Aperfeiçoamento de Pessoal de Nível Superior) and INC-SEC (Instituto Nacional de Ciência e Tecnologia em Sistemas Embarcados Críticos) for their support.

References

[1] A. S. Tanenbaum, Sistemas Operacionais: projeto e implementação, Bookman, 2000.
[2] J. O'Gorman, Operating Systems with Linux, Palgrave Macmillan, 2001.
[3] R. Souza Pinto, F. Monaco, Uma plataforma de aprendizado baseado em projeto para ensino e treinamento em sistemas operacionais, in: Anais do XXXII Congresso da Sociedade Brasileira de Computação, 2012.
[4] W. Christopher, S. Procter, T. Anderson, The Nachos instructional operating system, in: Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings, Usenix Association, 1993, p. 4.
[5] D. Holland, A. Lim, M. Seltzer, A new instructional operating system, in: Proceedings of the 33rd SIGCSE technical symposium on Computer science education, ACM, 2002, pp. 111–115.
[6] J. Carissimo, XINU–an easy to use prototype for an operating system course, ACM SIGCSE Bulletin 27 (4) (1995) 56.
[7] C. L. Anderson, M. Nguyen, A survey of contemporary instructional operating systems for use in undergraduate courses, J. Comput. Small Coll. 21 (1) (2005) 183–190.
[8] G. Fankhauser, C. Conrad, E. Zitzler, B. Plattner, Topsy–a teachable operating system, Computer Engineering and Networks Laboratory, ETH Zurich 2001.
[9] A. Tanenbaum, A UNIX clone with source code for operating systems courses, ACM SIGOPS Operating Systems Review 21 (1) (1987) 29.
[10] C. DiBona, S. Ockman, M. Stone, Open sources: Voices from the open source revolution, O'Reilly & Associates, Inc. Sebastopol, CA, USA, 1999.
[11]
[12] M. J. Bach, The design of the Unix Operating System, Prentice-Hall, 1986.
[13] R. Souza Pinto, Uma plataforma para ensino e treinamento em desenvolvimento de sistemas operacionais, Master's thesis, Universidade de São Paulo (June 2012).
URL `http://www.teses.usp.br/teses/disponiveis/55/55134/tde-06122012-105021/pt-br.php`
[14] A. Greg Kroah-Hartman, Jonathan Corbet, Who writes linux: How fast it is going, who is doing it, what they are doing, and who is sponsoring it: An august 2009 update., Tech. rep., Linux Foundation (August 2009).