Using a threaded framework to enable practical activities in Operating Systems courses

Aleardo Manacero, Renata Spolon Lobato
Department of Computer Science and Statistics
Paulista State University - UNESP
Rio Preto, Brazil
Email: aleardo@sjrp.unesp.br, renata@sjrp.unesp.br

***Abstract*—Teaching Operating Systems (OS) is a rather hard task, since being a OS designer is not a desired goal for most students and the subject demands a large amount of knowledge over system's details. To help this many courses are designed with laboratory practices, differing in how the practices are designed. Some try to implement next-to-real kernels, others use simulators, and even others use synthetic kernels. In this paper an approach based on synthetic kernels is described. It uses thread programming in order to establish control over the operating system components. This approach allows the kernel to grow following the materials presented in the course. It has been successfully applied in two different courses at our University, the first one being a basic OS course and the second one an upper level course. Results of these courses are presented.**

***Keywords*—Kernel implementation, Operating systems design courses, Operating systems laboratory, thread programming**

I. Introduction

One of the hardest subjects to be taught in computer science is the design of operating systems. Besides the existence of several textbooks covering the subject ([1], [2], [3], among other successful texts), learning these concepts involve knowledge in data structures, computer architecture and networking. Although learning this knowledge may be left to earlier prerequisite courses, it is not desirable to leave the OS course too late in the curriculum, which demands that part of that knowledge has to be brought inside the course. The consequence is that lots of material have to be covered, and understood by students, in order to have a complete view of the technologies used to implement an operating system.

To alleviate this situation most of the courses include laboratory practices. Since the subject is quite complex, different approaches have been suggested to tackle these practices. The most relevant are:

- **Kernel implementation**, where students have to implement an actual version of an OS kernel;
- **Simulators**, where students simply simulate the operation of kernel's components;
- **Component implementation**, where students implement certain components of a kernel, but do not connect them.

Each of these approaches have pros and cons, such as allowing in-depth knowledge of the system but demanding too much effort for the kernel implementation. One way to reduce the effort is to use real kernels and organize practices that are performed over these kernels, modifying or reimplementing some portions of it. One of such kernels is Minix [3], whose implementation has more than 27,000 lines. It is easy to imagine that such large code is hard to understand in a short period of time, even if only small parts of it are addressed.

Differently from the conventional approaches, the one proposed here involves the implementation of specific components, which are compounded through a kernel emulation framework based on threads. This allows for an incremental approach towards a complete synthetic kernel while maintaining a functional system along the way. Since the implementation can be incremental, the knowledge necessary is also incremental, enabling the offer of an OS course as early as the Sophomore year.

In the following pages one finds the description of this approach, followed by how it is deployed in two different courses offered to computer science majors. Results from these courses are presented, followed by a discussion about similar approaches. Conclusions from this experiment finalize the text.

II. Threaded Components of an OS

As previously stated, the practices intended to better understand the OS functionality are based in the implementation of its components through threads. In this section a full description about how these threads are implemented and how they are compounded in order to emulate an operating system is provided.

A. Compounding threaded components

A synthetic OS is created as a multithreaded system. This means that each of its components will be a separate thread, which is started in the occurrence of certain events in the system. This approach enables the implementation of most of the services provided by the OS as an independent component, that can be executed without the implementation of the remaining system.

Figure 1 shows the main structure for the synthetic kernel. As one can see, the program has an infinite loop over a function (control_unit) that returns the event that must be managed in each loop. Its output defines which thread must be started, that is, which OS component will act in that cycle. Each component is implemented by a single thread, in order to allow effective concurrency among them. c

```
OS_Kernel {
    Boot_init();
    while (true) {
        op = control_unit();
        switch (op) {
            case 0 : start thread A; break;
            case 1 : start thread B; break;
                     .
                     .
                     .
            case n : start thread N; break;
        }
        update_status();
    }
}
```

Fig. 1. Framework for an Operating System synthetic kernel

After starting a thread the kernel proceeds updating the system status, what includes presenting the actions to the user. This is performed by the update_status function. This function has also to schedule the next process to the CPU, based in what was the triggered event and the previous status of the system. It must be noted that the scheduler, although part of the process management, is not treated as a thread, since it has to provide a result before any other event can occur.

With this framework the instructor and students get a fully functional system, even if only few components are implemented. The parts that must be implemented are the two external functions in the framework (control_unit and update_status), the process table structures, and the desired OS components. The control_unit may be implemented through simple specifications, such as a random generator or as an input interface. The update_status can be implemented as an interface to show the system's status and eventual manipulations over the process table. It is possible for the instructor to either provide both functions in order to guarantee a customization over the kernel's inputs and outputs or request their implementation from students.

All OS components can be specified and implemented independently. They are synchronized through the access to the process table, which provides the necessary data structures to manipulate the running synthetic processes. The interactions between components are performed by calls to specific threads.

The main advantage of this approach is that the instructor can pick only the components that he/she thinks that are more important, reducing the amount of implementation needed. Another advantage is that the amount of details in each component can also be controlled by the instructor. This way students can get a hands-on practice over important components of a OS, without the burden of knowing complex details about hardware or even understanding a huge amount of code written by someone else.

B. Creating specific threaded components

One should have noticed that the external functions and the process table are vital parts of this approach. However, their discussion will be left to the next section, where the application of this approach in two different courses is described. Here, we are concerned only with the specification of threaded components, since they are the core of students work even if they have to implement the whole framework.

As said before, the instructor can choose which OS components will be implemented. Surely, some components are essential, such as the process management control, but each functionality can be implemented, and verified, independently. In order to enable such scheme each thread has to be modeled as an independent piece, that access the shared process table. Therefore, each thread has to:

- Access the process table in a mutual-exclusion approach;
- Perform the necessary process table manipulation, such as blocking or releasing a process;
- Call, when necessary, the process scheduler or other OS threads.

These requisites are easily achievable. This happens because the amount of interactions between threads can be increased incrementally, starting with very simple components before proceeding to more complex ones. However, even operations much more complex such as managing virtual memory, can be easily implemented with a careful design of interactions between memory management and disk and processes management components.

III. Application of the Threaded Kernel in Specific Courses

The threaded approach just presented has been applied in two different courses offered to Computer Science majors. The first course, in which this approach has been applied since 2007, is an introductory Computer Systems course titled Foundations of Computer Systems which covers operating systems and computers networks in a single course [4]. The second course, in which the application occurred since 2009, is an advanced course titled Operating Systems Design, which is mandatory only for students following the Computer Systems track inside their major[1]. The application of this approach, including examples for each course, will be described in separated sections.

A. Threads in the Foundations of Computer Systems course

This is a mandatory course for students pursuing the computer science major. It is intended for students in their 4th semester in the university with knowledge in data structures and digital circuits. Since it is also an introductory course, covering a quite large amount of subjects, there is not much time available for laboratory practices. The syllabus covers these major topics:

1) Interaction between OS and networks to establish distributed and parallel systems;
2) Operating systems management components, including an introduction to concurrent programming;

[1]The Computer Science major is split into four tracks after sophomore year (Computer Systems, Information Systems, Digital Control and Automation, and Scientific Computing), and students have to chose one of them to get a degree [5].

```
TEMPO DA CPU: 11

MENSAGENS DA THREAD ioFinish:
HOUVE UMA CHAMADA PARA FINALIZACAO DE IO.
NO ENTANTO, NAO HA PROCESSO BLOQUEADO NO BCP.
LOGO, NAO PODE HAVER DESBLOQUEIO.

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::           DADOS SOBRE O BCP                          :::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: NP    Estado                 TE      IO      Pr          ::
:: 1     ::::ATIVO::::          2       1       3           ::
::                                                          ::
:: LEGENDA:                                                 ::
:: NP: NUMERO DO PROCESSO                                   ::
:: TE: TEMPO DE EXECUCAO                                    ::
:: IO: NUMERO DE ENTRADAS E SAIDAS                          ::
:: Pr: PRIORIDADE DO PROCESSO                               ::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

Fig. 2. Output produced during the execution of a student implementation

3) Computer networks protocol (RM-OSI and a brief overview of TCP-IP);
4) The flow goes back and forth from OS to networks contents in order to show their relationships;
5) A thorough description of this course, at its origin, can be found in [4].

The fact that it is mandatory and cover many topics is the main motivation to use an incremental approach for practices such as the one presented here. Since there is not much time available, only few components are actually implemented here. Our choice usually falls into implementing the process management, including the dispatcher, and the disk I/O management.

As a side note, assignments approaching computer networks include isolated tasks, such as the implementation of a sliding-windows protocol emulation, the spanning-tree algorithm, and the ARP protocol among others.

It is possible to ensure a challenging task each year, even asking for the same components, if the algorithms for dispatching and disk scheduling are different every year. Changing these algorithms implies in modifications on how processes are managed or interact, avoiding that the practices could become repetitive. For example, in the most recent offer of this course students had to implement a priority-based dispatcher, which demands that information about the processes priority have to be stored in the process table. Dispatchers implemented in previous years included round-robin, SRTF (Shortest-Remaining Time First), and an I/O bounded algorithm.

Each practice (implementation) is evaluated accordingly to three rubrics: program correctness, code comments, and output interface. The first two are quite obvious and do not demand further description. The output interface is judged considering the amount of organized data is provided about the "execution" of processes in the multithreaded operating system being emulated, that is, the interface has to provide clear data to follow the program execution and check its correctness. Therefore, a graphical interface is not required and Figure 2 shows an output of a student program, during emulation.

Although the text in that figure is in Portuguese, some remarks can be drawn from it. First, it provides a system's clock in its top line, followed by a message that tells which

```
ID      PRIORIDADE      STATUS
6       3               Execução
7       3               Pronto
9       3               Pronto
8       1               Bloqueado
10      2               Bloqueado

Endereço de Memória     Trilha  Bloco   ID      ProcessoStatus  Requisição
        4987            156     27      10      Esperando E/S   Leitura
        2686            84      30      10      Esperando E/S   Leitura
        860             27      28      10      Esperando E/S   Leitura
        1810            57      18      8       Esperando E/S   Escrita
        3055            96      15      8       Esperando E/S   Leitura
        1944            61      24      8       Esperando E/S   Leitura
        1596            50      28      8       Esperando E/S   Leitura
        3386            106     26      0       Executando E/S  Leitura
```

Fig. 3. Output produced during the execution of a second practice implementation, now including disk access requests

event will be treated (the conclusion of an I/O operation in this case), and how to react to it (doing nothing in this case since there was no process waiting for I/O[2]). Then, it presents a process table, with information about all processes executing, where the five columns indicate the process identification (NP), status (Estado), CPU time already used (TE), number of I/O operations performed (IO), and the process priority (Pr). Therefore, it presents enough data to check the system execution, step-by-step. As a side note, a random generator is used to generate the events in the system.

Figure 3 presents the interface from a different student, showing the execution of the same system, but now including the management of requests for disk access using the scan algorithm. In this interface the student presents a list of processes "running" in the system, in a tabular format containing the process ID, its priority and status (considering three possible states for a process, that is, it can be allocated to the CPU - "Execução", in the ready queue - "Pronto", or blocked - "Bloqueado"). After this it shows the list of disk I/O requests, showing the memory and disk locations, the process that requested the operation, operation status and type (input or output).

Each assignment has to be finished in a period lasting from 15 to 20 days and is executed by pairs of students. The first assignment usually takes more time since students have to implement not only the process manager but also several utility components. They also have to get acquainted with thread programming with C language.

Since its first application, this approach for lab practices has improved the understanding of operating system concepts. Students get better prepared for written evaluations and the failure rate decreased by a large margin. Statistical data available shows that the failure rate was reduced threefold (from around 30% to 10% for classes were this approach was used).

Tables I and II show the results from a survey applied for the most recent class, which concluded in December 2012. From a total of 15 pairs of students we received answers from 10 pairs (66.67% of the total), using Google Docs forms.

From table I we can see that students had more trouble with the assignments related to OS, which confirms the notion that this is a difficult subject. It must be noted also, that they

[2]It should be noted that this specific situation occurred because the random generator does not check which events are feasible at any given time, what is not a correctness problem.

TABLE I. SURVEY RESULTS ABOUT DIFFICULTY OF DESIGN PRACTICES IN THE FOUNDATIONS COURSE (% OF RESPONDENTS)

	Very hard	Hard	Median	Easy
Overall difficulty	10	30	45	15
First assignment (process management)	30	70	0	0
Second assignment (I/O management)	0	80	20	0
Average of the other assignments (computer network protocols)	0	25	40	35

TABLE II. SURVEY RESULTS ABOUT USEFULNESS OF DESIGN PRACTICES IN THE FOUNDATIONS COURSE (% OF RESPONDENTS)

	Very useful	Useful	Somewhat useful	Not useful
Overall usefulness	40	30	30	0
First assignment usefulness	40	30	20	10
Second assignment usefulness	40	30	30	0
Other assignments usefulness (computer network protocols)	35	25	30	10

felt more comfortable with the second assignment, when they already knew the framework of the emulated kernel.

Although not presented in this table, some other aspects collected from the survey must be listed here. In the first assignment students revealed that their major difficulties were related to thread programming and the understanding about what had to be implemented. In the second assignment, their difficulties were related to the I/O mechanisms and the relationship between the kernel components. Another complaint was related to the specification of the assignments, which some students had trouble in identifying the actions that their systems had to perform.

From table II it is possible to notice that students have the feeling that they learned better by implementing the components that were required in the practices. It is also possible to identify that they felt that the OS practices were a little more useful (70% of respondents) than the computer network practices (60%). It is important to notice also that this feeling could be verified in the students' grades, which improved reasonably from classes where there was no laboratory assignments.

B. Threads in the Operating Systems Design course

The same approach has been applied in a more advanced course, which is taken only by students that are in the Computer Systems track. This means that the classes are smaller and that more material can be covered in the practices. This course can be divided in two disjoint parts:

1) A theoretical view of distributed operating systems and distributed systems, including synchronization algorithms, fault tolerance and replication;
2) A design view of conventional (single processor, multiuser, multitask) operating systems, based in the threaded kernel introduced in the previous course.

Since students enrolled here have already took the Foundations of Computer Systems course, they know in advance the structure of the threaded kernel used in the practices. This makes easier to develop each component further. Therefore,

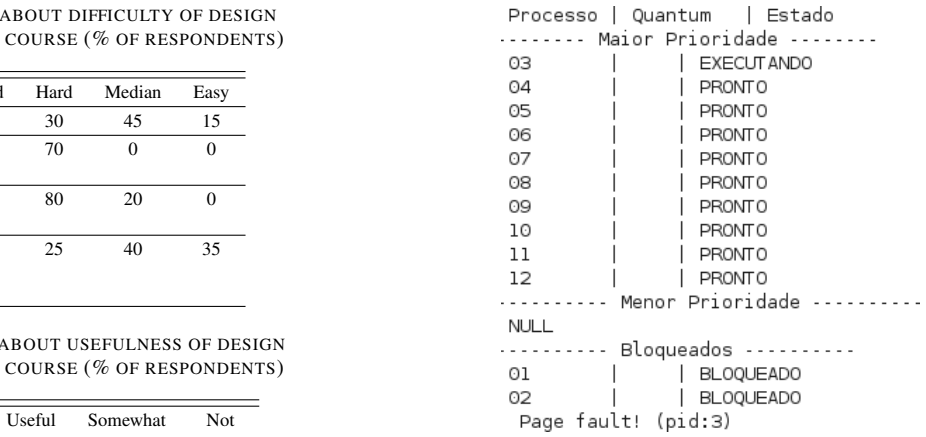

Fig. 4. Snapshot of an OS emulation running memory management components

the design practices involve more components with a higher degree of interaction between them. As an example, Figure 4 presents a snapshot of executing processes with management of page misses and virtual memory, which are functions usually not implemented in the first course. In this figure we can see in the last line a message saying that the process with "pid=3" caused a page fault. In the top part of the figure this process appears as running (EXECUTANDO), and the fault will move it to the blocked processes list (Bloqueados), that contained process 2, just before the fault.

To facilitate experimentation and to allow a better understanding of the whole kernel, in this course the control_unit() function is implemented as a reader of synthetic programs. These programs use a simple syntax to map execution events in regular programs. The control_unit() function creates a "process control block - PCB" where each process has a pointer to a file containing the synthetic program. It emulates its execution by reading one line and applying the correspondent action until the end of the file.

Figure 5 shows a short example of a synthetic program. In its header (first five lines) we find the program's name, file identifier, original priority, file size, and a list of semaphores used by the program. After that comes the synthetic code, where the instructions "read x" and "write x" are disk access requests for the track x, "exec y" means CPU processing during y time units, "V(z)" and "P(z)" are calls to operations over semaphore z. Other synthetic instructions may include I/O operations to specific devices, e.g. printers or monitors, and process creation through fork.

With the synthetic commands we can represent a large amount of events that occur in a conventional system. The threads for each OS component can be specified including more complex functions, such as manipulating virtual memory, page tables and so on, as indicated in the example in figure 4. Additionally, since synthetic programs comprise what could be hundreds of bytes in a single command, page manipulation is made by establishing that each page can accommodate k synthetic commands, and that the memory has N pages available to the processes. This allows the implementation of paging mechanisms, including the treatment of page faults and address translation.

```
sint1
1
1
32
s t

exec 4000
read 20
exec 500
read 30
exec 5000
P(s)
exec 200
P(t)
exec 10000
write 20
V(t)
V(s)
exec 2500
```

Fig. 5. Example of a synthetic program ran by the threaded kernel

The use of synthetic programs make it easier to simulate the OS operations. Students can concentrate in them, instead of how interruptions and system calls could be generated. Although this approach could make easier the practices in the first course, we do not use synthetic programs there in order to ensure a better understanding of the hardware operation. We believe that this is important because most of the students will not take the OS Design course, having only that opportunity to be in contact with such issues. Therefore, making they work extra on managing interrupts and syscalls is a valuable effort.

We do not have a quantitative evaluation about the application of this approach in the OS Design course. However, we have qualitative insights from students that took this course in previous years, and they do not differ from those provided by the surveys applied in the Foundations of Computer Systems course. This should not be surprising since students in the advanced course are a subset of students that took the first course.

IV. Related work

As previously stated, the practices in operating system courses follow three different approaches. Here we will further discuss only the approaches based in simulation and component implementation, while a broader and more detailed review can be found in [6].

An approach based in component implementation is presented by Laadan, Nieh and Viennot [7], where they use the Linux kernel as a testbed for modifications in specific components, such as system calls or the scheduler. This is actually a continuing effort on previous works from the same group. Their approach demands, however, a large knowledge about Linux implementation.

In other direction we find the work from Robbins [8], where students can practice disk scheduling algorithms in a simulator built with this purpose. Although it is very useful for this topic, it does not address other important aspects of an OS and also do not demand any implementation from students, being simply a help for the understanding of the algorithms.

Nachos [9], is a traditional simulator used as a teaching operating system, replaced by Pintos [10]. Pintos is a full machine simulator and students are required to implement/modify small parts of it. Although several components can be targeted it is hard to understand the whole OS operation from these parts.

Finally, we have PennOS [11], where a user-level OS is simulated through the use of the *user context* library in order to enable low-level implementation of OS mechanisms without the need of special access to hardware instructions. This demands a strong knowledge about this library and low-level programming, as indicated by students reviews.

V. Conclusions

In this paper we described a multithreaded approach for laboratory practices in Operating Systems courses. This approach has been applied into two different courses taken by computer science major students, using different levels of components.

From the reports given by students, including answers to the survey applied in the intermediate level course, we can conclude that the use of threaded components allowed for:

- A better understanding of the concepts involved with the implementation and operation of an operating system;
- An improvement in the average grades in both courses in about 10%, with the rate of students failing to pass the first course dropping from an average of 30% from 1998 to 2006 to an average of 18% since then;
- A reduction in the number of students avoiding the Computer Systems track, associated with an increase in the number of students doing their graduation papers in topics related to distributed computing, from 4-6 students per year to 8-10 students nowadays.

Besides the success achieved, there still some aspects that can be improved, such as:

- Establishment of different command architectures for the language used in synthetic programs, which would allow for their use even for a very small set of OS components;
- Creation of a common framework between the simulated components and user interface, reducing the number of components that have to be implemented by students;
- Improvement in the specification of all assignments, especially the first assignment in the Foundations course, since it is in that moment that the whole framework is defined.

Therefore, our final remark is that this approach for OS practices can be very useful, since it provide a deeper understanding about key components of an OS at the same time it does not requires a large knowledge about the kernel or even programming details.

ACKNOWLEDGMENT

The authors want to acknowledge their gratitude with the many students that took the courses presented here. Without their readiness to take part in the practices and to learn the effective use of multithreaded programming, it would be impossible to achieve the results presented here.

REFERENCES

[1] A. Silberschatz, P. Galvin, and G. Gagnon, *Operating Systems Concepts*, 8th ed. John Wiley, 2011.

[2] W. Stallings, *Operating Systems: Internals and Design Principles*, 7th ed. Prentice Hall, 2011.

[3] A. Tanenbaum and A. Woodhull, *Operating Systems Design and Implementation*, 3rd ed. Prentice Hall, 2006.

[4] A. Manacero Jr., "Merging operating systems and computer networks: why and how," in *Proc. of the International Conference on Engineering Education, ICEE98*, 1998.

[5] A. Manacero, R. dos Santos, N. Marranghello, A. Pereira, A. Cansian, and J. Ralha, "A flexible curriculum for computer science undergraduate major," in *Frontiers in Education Conference, 2001. 31st Annual*, vol. 2, 2001, pp. F3D–20–5 vol.2.

[6] C. L. Anderson and M. Nguyen, "A survey of contemporary instructional operating systems for use in undergraduate courses," *J. Comput. Sci. Coll.*, vol. 21, no. 1, pp. 183–190, Oct. 2005.

[7] O. Laadan, J. Nieh, and N. Viennot, "Structured linux kernel projects for teaching operating systems concepts," in *SIGCSE*, 2011, pp. 287–292.

[8] S. Robbins, "A disk head scheduling simulator," in *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, ser. SIGCSE '04. New York, NY, USA: ACM, 2004, pp. 325–329.

[9] W. A. Christopher, S. J. Procter, and T. E. Anderson, "The nachos instructional operating system," in *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, ser. USENIX'93. Berkeley, CA, USA: USENIX Association, 1993, pp. 4–4.

[10] B. Pfaff, A. Romano, and G. Back, "The pintos instructional operating system kernel," in *Proceedings of the 40th ACM technical symposium on Computer science education*, ser. SIGCSE '09. New York, NY, USA: ACM, 2009, pp. 453–457.

[11] A. J. Aviv, V. Mannino, T. Owlarn, S. Shannin, K. Xu, and B. T. Loo, "Experiences in teaching an educational user-level operating systems implementation project," *SIGOPS Oper. Syst. Rev.*, vol. 46, no. 2, pp. 80–86, Jul. 2012.