Teaching Operating Systems Concepts with Execution Visualization

Abstract

We present an original approach to introduce Operating Systems concepts to Computer Engineering undergraduate students. These concepts are the basis on which students build a mental model of the whole computer in order to make important design decisions throughout their career. One major challenge in teaching operating systems is the complex, intangible, and nondeterministic nature of an actual computer system containing many cores operating in parallel.

We propose a global approach to address this challenge involving a full-scale open source operating system, a carefully designed set of experiments and novel execution visualization tools. In order to deconstruct their preconceptions, students are exposed to phenomena that seem contradictory at first glance, but are the result of the interaction between the microarchitecture, the operating system and the libraries. In the spirit of constructivism, students are invited to observe the effect of running their own programs as part of a problem solving challenge. Participants can thus observe the duration of underlying system calls and the actual scheduling performed by the operating system which is otherwise hidden. Experiments are proposed to compare the impact of design choices and to lead to improved awareness of performance implications. We describe five problem solving activities that we developed and expose the purpose of each tool used. In the context of the first semester of deployment, we evaluated the activities using a qualitative method. We conducted online surveys and a focus group, and observed a high learning satisfaction level for students. This validates the proposed approach with a high level of confidence.

1 Introduction

The operating systems course is part of the classical curriculum of undergraduates in software and computer engineering. The content is well established from decades of iterative refinement and covers topics such as task management, system calls, synchronization, scheduling, memory management, and file system structure.[1,2] There are usually practical activities, or laboratory assignments, that complement the lectures. One approach to these activities consists in using simulators.[3–5] A simulator can help visualize the execution of classical algorithms, step-by-step. However, subsystems interaction is not covered from these activities, such that the global perspective is missing. The other common approach involves programming a small scale operating system[6,7] or modules of an existing operating system.[8] Unfortunately, this approach does not recognize that most students will use, rather than develop, an operating system, and that making one does not automatically translate into using them efficiently. Both the simulation and implementation approaches lack the time dimension of an actual system. As such, being a good auto mechanic does not necessarily

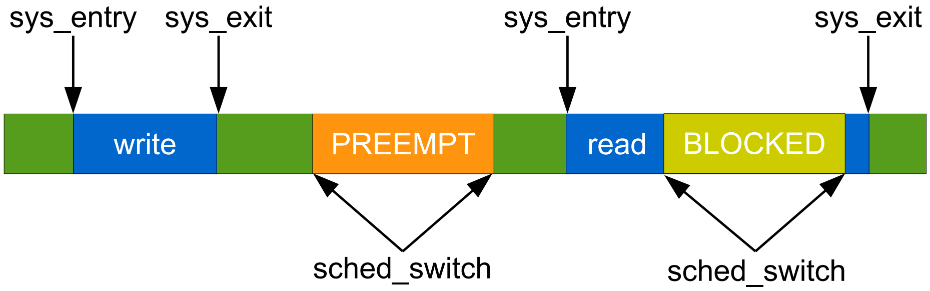


Figure 1: Control Flow View example

make you a good racing driver, although both require a thorough knowledge of the car and its parts. In addition, students are often reluctant to work with a toy like small-scale operating system, not in widespread use.

Instead, we propose to use an actual operating system implementation and visualize its execution by tracing it. Visualization tools evolved since early attempts to use them for teaching,[9] creating an opportunity to increase comprehension by integrating these tools. Our hypothesis is that this approach, based on observation, is well suited for an introduction to operating systems.

Our contribution consists in designing activities involving execution visualization tools. Among them, the main one is the control flow view. This view displays the process state as a function of time. Figure 1 shows an example, including key events of a running task. The process runs in userspace and then performs a write. The task is preempted, either because a more important task becomes ready or the task quantum expired. The rest of the execution shows a blocking read, that yields the CPU while data is loaded.

The proposed activities are based on a constructivism background,[10] a well established pedagogical perspective in engineering education.

2 Course overview

The targeted operating system course is part of the first year of specialization. It has both the computer organization and object-oriented programming courses as prerequisites. It is important to realize that the presented activities are designed in that context, because the constructivism approach, on which they are based, requires that the participant should have proper prior knowledge to be effective.

Activities involves mainly C/C++ code, while some small parts are in assembly. These programming languages are typically used for their ability to reveal the hardware and software interfaces. In the present context, this choice also has the advantage of simplifying the relationship between the code and it's run-time representation at the system level. We highlight the pedagogical value of each reference to assembly in activities presentations.

The first four activities are performed on Linux workstations running the current Fedora release. We selected Eclipse CDT as IDE for its convenience and availability.[11] We used the Linux Trace Toolkit next generation (LTTng) as kernel tracer, in addition to the User-Space Tracer (UST).[12] This combination allows correlating events between the operating system and the application. We used the Eclipse Linux Tools Tracing and Monitoring Framework as trace viewer for its ease of use, and also because students were already familiar with it. We also used other standard tools such as `strace` and `hexdump`.

We analyzed the security implications of kernel tracing before the installation of LTTng. The IT department does not grant root access on workstations, as required to record kernel events with perf.[13] Fortunately, LTTng can be configured such that a group can trace the kernel without requiring additional privileges. Since the tracing session records events for the whole system, we verified that the instrumentation available does not leak sensitive information. In particular, we verified that passwords can't be recovered from keyboard interrupts and that user-space memory accessed by system calls is not recorded. We considered that recording opened files names and command names without arguments is not an issue, since few restrictions exist to access this information in the first place.

The fifth and last activity focuses on the Windows API. Technically, it would have been possible to do this activity on Linux by cross-compiling and running it with Wine.[14] Paradoxically, it may represent a compelling approach to study the internals of the Windows API. However, we opted instead for the simplicity of using Windows workstations during the first iteration. We nevertheless applied the same spirit of run-time analysis with a trivial tracing implementation.

3 Activities description

In this section, we present activities and explain the intent behind each one in terms of pedagogical objectives. Activities are developed according to the following cognitive strategies:

- Comparative study: compare related concepts in order to reveal similarities and differences. It aims at building the student's knowledge network.
- Parameter exploration: follows a scientific approach of changing one parameter value to see its effect.
- Exposure to surprising phenomena: challenge preconception and foster questioning that prepare students to actively search for answers.
- Problem solving: practice newly acquired knowledge by achieving programming challenges inspired from actual situations.
- Self-verification: test cases that the student can run to verify that its implementation produce the expected result. Aims at increasing student's autonomy and confidence.

We use execution visualization in each activity as a mean to activate these cognitive strategies. The visual representation allows the construction of an abstract mental model of the system, including the time dimension. In addition, because the system interface is a convention determined by the hardware, that every programming language must respect, the tracing view is a strong generalization that helps to create rich connections between concepts of each component of the system. Whenever possible, we tried to link activities to real situations that students can relate to. Thus, participants can appreciate the importance of their learning while developing their programming skills.

Each session lasts three hours. We expect students to spend a few additional hours to complete their analysis and write a short report, following each session. We think that the proposed activities are very intense in terms of amount of material, experimentation, and code, and therefore use three strategies in order to save time. First, we provide a code skeleton containing boilerplate code that does not add pedagogical value. Participants can free their mind from gory details and focus on the specific aspect that matters. Second, we provide scripts to automate repetitive steps, such as launching tracing experiments. Third, the instruction document serves also as a report template, thus allowing students to write findings while performing the activity. This document is part of a paperless grading process, and is returned annotated. Because the setup is well established, we can afford to review their code and provide valuable feedback.

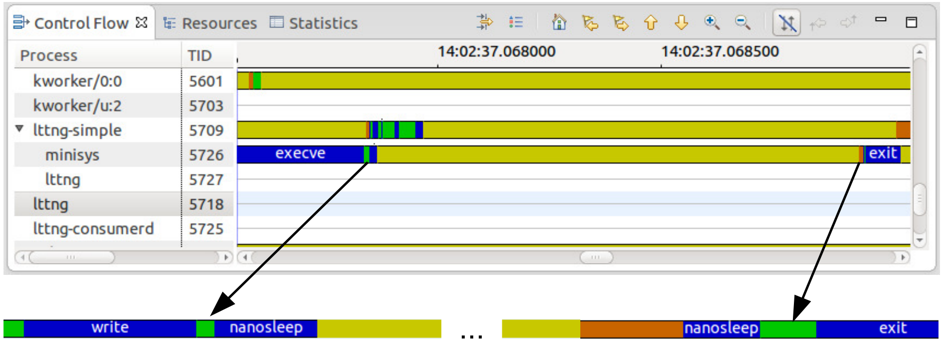
3.1 Session 1: Introduction to concepts

Students have programmed a robot on an embedded board as assignment in a previous semester project course. We take advantage of this prior knowledge as a starting point for the introduction. The teacher deliberately introduces two bugs in a program that controls LEDs, to demonstrate inherent limitations of this bare-metal platform lacking an operating system. The first bug makes the program enter an infinite loop that puts the system in a non-responsive state. The only way to recover from that fault on this system is to press the reset button. We then enable a routine that overwrites the output pins, making LEDs behave abnormally. This example helps highlight the fact that there is no way to restrict access to output ports, and even to bus addresses in general. This demo aims at raising questions about the purpose of the operating system in terms of reliability and security.

The core activity consists in evaluating the precision of `nanosleep`. We chose `nanosleep` for its blocking behaviour that involves the scheduler in a deterministic way. Programming in assembly at this stage is essential to reveal the purpose and the necessity of the `syscall` instruction of the processor. A system call, as explained in the course lectures, triggers the request to the operating system and consequently allows privilege separation. The student can then realize that his code stops executing while the system call is performed. Programming in assembly makes sure there is a one-to-one relation between the code and the actual execution. The equivalent C program, linked to the standard library, produces numerous non-related system calls for library loading and memory management, and we want to avoid the confusion brought by these potentially overwhelming details. The assembly program

```
$ strace ./minisys > /dev/null
execve("./minisys", ["./minisys"], [/* 54 vars */]) = 0
write(1, "\nHello!\n", 8)                     = 8
nanosleep({0, 1000000}, NULL)                 = 0
_exit(0)                                      = ?
```

Figure 2: System calls performed by the program



Process	TID
kworker/0:0	5601
kworker/u:2	5703
▼ lttng-simple	5709
minisys	5726
lttng	5727
lttng	5718
lttng-consumerd	5725

Figure 3: Process state of the program according to time

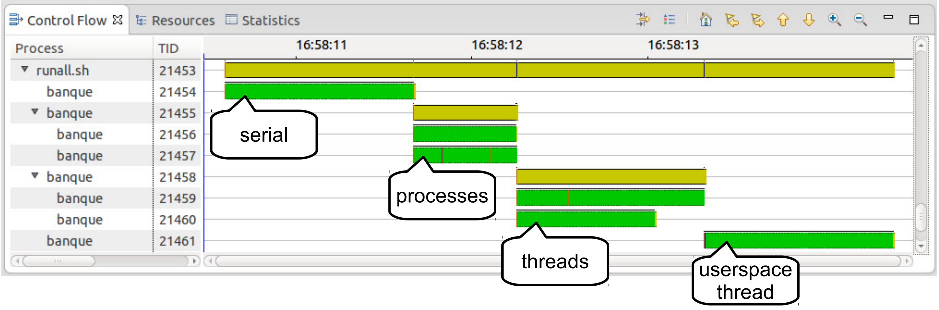
that we provide already contains one write and the final exit system calls. It also includes the calling convention to follow and a link to the system call declarations, directly into the source code of the Linux operating system.

Students can then verify their program by running it with `strace`. The resulting output in Figure 2 shows each system call actually performed, and thus can be compared with the expected system calls.

Once the program is verified, a kernel trace of the program execution is created and opened with the Eclipse trace viewer. The result from the control flow view is shown in figure 3. Convenient information is displayed, like system call names and intervals duration (as mouseover). Students can explore the execution and compare various delays in the system, such as the time required to perform the write and the actual sleep time of the nanosleep. It becomes clear that `nanosleep` is a type of passive waiting that yields the CPU, involving the scheduler.

3.2 Session 2: Processes and threads

The objective of the second activity is to present ways to spawn execution units and run new programs. The session is divided into three parts. The first part concerns the difference between types of execution units (processes and threads). The second part consists in an introduction to `exec()`, and the last part is an exercise combining `fork` and `exec`.



Process	TID
runall.sh	21453
banque	21454
banque	21455
banque	21456
banque	21457
banque	21458
banque	21459
banque	21460
banque	21461

Figure 4: ATMs experiments in the control flow view

The session starts by a simulation of two ATMs on a global bank account, each doing operations simultaneously. The serial code is provided and the code to spawn virtual ATMs must be completed using processes, normal threads, and userspace threads. The effect of these operations on the final balance is observed. When using processes, the final balance does not change despite operations in the spawned processes. When using normal threads, the balance computed is incorrect and differs from one run to the next. Only when using userspace threads is the right balance computed. Our intent is to expose students to strange phenomena in order to raise questions and put them in a state where they will actively seek an explanation to these phenomena.

The program is then traced with LTTng, as shown in figure 4. By zooming on the locations where tasks are spawned, the student sees that a call to `clone` is done to create either processes or threads, and thus that the difference resides in the arguments provided to this system call. LTTng doesn't decode flags passed to the `clone` system call. The student uses `strace` to decode these arguments. We explain how to access the documentation about system calls with `man` so that students can search for information about relevant flags which describe the effects on address space sharing and explain their differences. In the case of userspace threads, there is no `clone` system call performed and this execution looks like the serial one, therefore explaining why no race condition occurs.

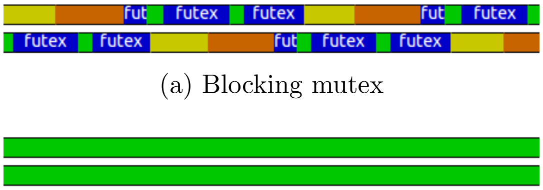
The next activity is a small challenge that consists in executing three programs with `exec`, one after the other, without creating new processes. The root program has three calls to `exec`, but only the first one is executed, and the rest of the program is not executed as one would normally expect. This is done to underline a key aspect of `exec`, namely that it does not return on success, and to challenge a preconception about program execution.

The last activity of this session is to create a wrapper that executes a program without address randomization, a feature used to make security attacks harder. In some circumstances, one may indeed need to disable this feature, in particular when studying security attacks themselves. Students need to complete the code of the wrapper by combining `fork` and `exec`. We provide a program that displays key addresses of variables and functions in order to observe the effect of the randomization and to allow students to test their implementation.

3.3 Session 3: Synchronization, signals and inter-process communication

The session starts with a comparison between three types of locks, used to serialize concurrent operations on a structure shared by two threads. For each execution, we can specify the length of the critical section and the number of cycles to perform. The critical section should be protected by these three locks. The first type of lock consists in a single mutex, equivalent to a binary semaphore. The second type of lock consists in a set of relayed binary semaphores, one per thread. The third and last type of lock studied consists in a minimal spinlock implementation that we provide in assembly. The program includes a check command that allows students to verify the proper working of locks.

The program is then run under tracing. Various observations can be made concerning performance issues, fairness, reliance on system calls, and blocking behaviour of the different types of locks. For instance, Figure 5 a) shows that a mutex is implemented with the `futex` system call, that can block, while spinlocks hog the CPU in userspace. This can be explained by looking at the spinlock implementation in assembly where it becomes obvious that CPU cycles are burned while polling for the lock. Another example, about lock fairness, is given in Figure 6. We observe in a) that a mutex may be unfair if the lock is obtained just after being released, while chained semaphores in b) produce a pattern that resembles a checkerboard. However, if the critical section is short and frequent, a simple mutex is much faster on average than the perfect fairness achieved with chained semaphores, where a context switch is forced at each cycle. Understanding the fine differences between lock types, clearly exposed by looking at their detailed behavior, would be hard to achieve without tracing.



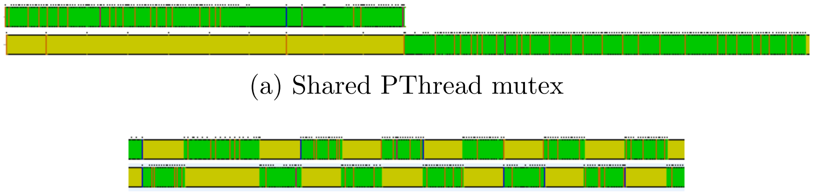
(a) Blocking mutex

(b) Spinlock running in userspace

Figure 5: Difference between mutex and spinlock

The next activity is about implementing a deadlock detection algorithm between two threads. Students already know from the course material that a deadlock can occur when two mutexes are locked in reverse order. The deadlock can be observed when the output of the program stops and must be terminated by hand. We ask students to automatically exit if a deadlock occurs. A timer signal is configured to periodically call the `watchdog` method, in which the deadlock detection must be implemented. Signal `SIGALRM` will interrupt the program execution by nesting over the signal handler even if the application is deadlocked.

The last activity concerns a producer-consumer application communicating with pipes. The goal is to compute word frequencies in a text. The first process tokenizes words read from



(a) Shared PThread mutex

(b) Chained semaphore

Figure 6: Fairness to access a critical section

standard input and writes each resulting string and their length on their own pipe. The second process counts word frequencies in a hash map. The application stops when there is no more input to process. Figure 7 shows the architecture of the application. The tokenizer and the frequency code is provided, and students have to connect the two processes with pipes. One drawback of the default signal handling is that all tasks quit and partial results are lost when control-C is hit. The last task is to overload signal `SIGINT` in order to stop processing and to display partial results before exiting.

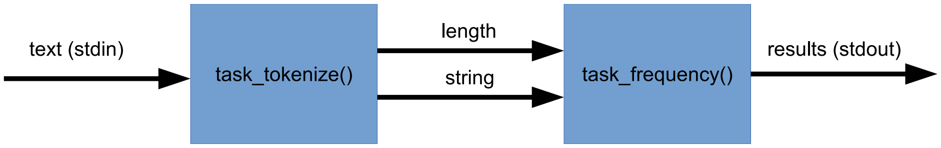


Figure 7: Architecture of the word frequency app

3.4 Session 4: Memory management

The first activity of this session is to experiment with conditions that raise segmentation faults. Students know that a segmentation fault is some kind of invalid memory access, but what really triggers them? To answer this question, the memory is scanned from a valid address until a segmentation fault occurs. The signal handler for `SIGSEGV` is overloaded to display the last scanned address, together with its offset from the start address, before the program exits. The scan is performed forward and backward to locate the data address range within the valid address range. In addition, the program saves the `/proc` file representing the virtual memory area maps of the process. The students now have all elements in hand to connect these concepts, since the scan yields addresses from the maps.

But what does the content of a memory page look like? To answer this question we propose to dump a whole page from the stack and the heap into a file. We write fun hexadecimal patterns to memory as distinct marks such as `0xCAFEBABE` and `0xDEADBEEF`. The code that saves the page must be completed. The main task here is to compute the start address of the page from the pointer. Once pages are saved, students can use `hexdump` to display the raw memory content and find the location of the marks. These findings are used to explain the result of the next activity.

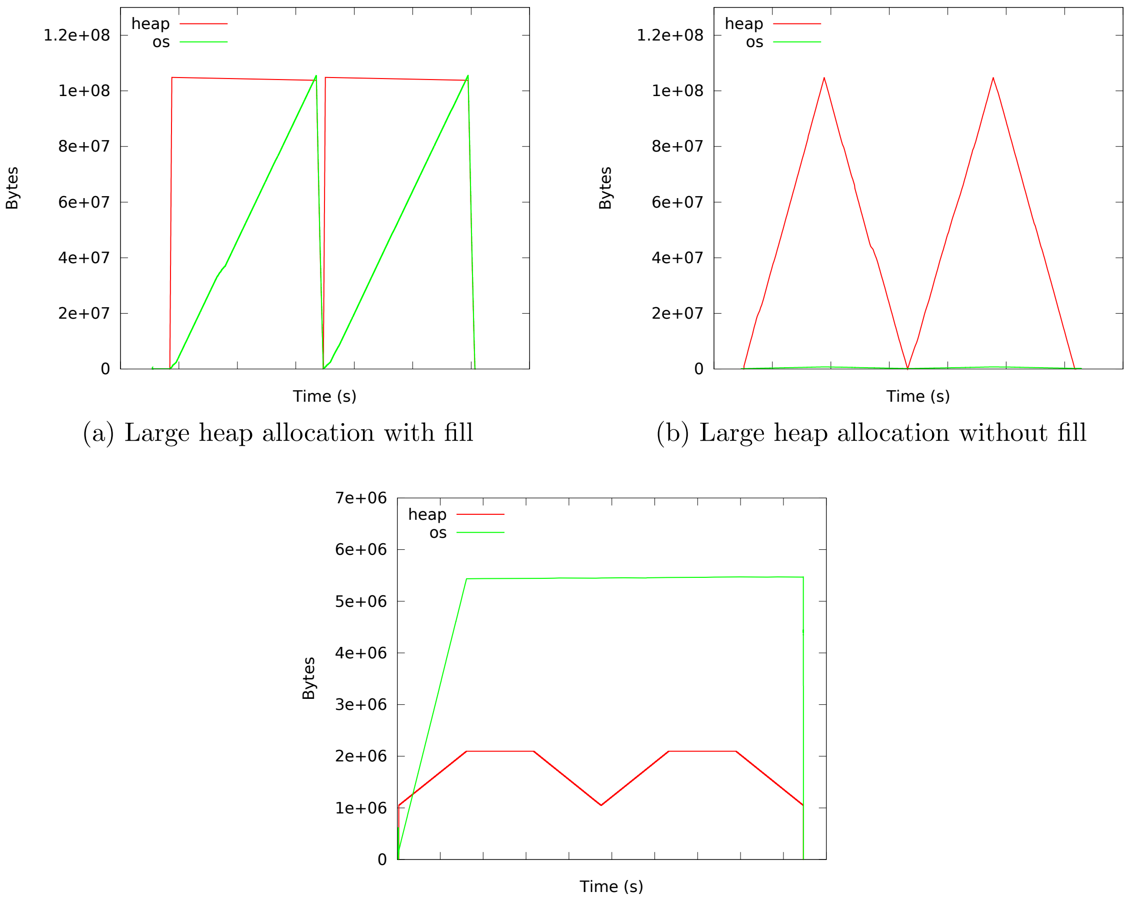
The last activity of this session involves tracing the memory demands of an application and relating it to pages effectively allocated at the system level. Three event types are recorded. First, the stack size is measured by saving the stack pointer on each function entry. Second, heap memory allocations are tracked by tracing calls to `malloc()` and `free()`. Last, kernel events `mm_page_alloc` and `mm_page_free` indicate the amount of pages mapped to the process address space. We provide the application `drmem` that makes allocations on the heap or the stack. Many parameters can be specified like the number and size of allocations, the number of allocation and deallocation cycles, and whether to access or not the allocated memory. Figure 8 shows the amount of memory allocated with respect to time for three heap experiments. In a), large chunks of one megabyte are allocated on the heap, but not filled. In this situation, almost no page is effectively allocated by the operating system. In b), the same experiment is done, except that the memory allocated is actually filled. We see that pages are allocated as the memory is filled, and this is related to the page fault mechanism. The third experiment in c) involves allocating one million integers, but then the number of pages ends up greater than the usable memory. The alignment and metadata maintained by the allocator explains this overhead and is clearly visible from the page dump of the previous experiment. In these three experiments, the amount of memory allocated by the operating system may be below, equal, or above what is requested. It aims at presenting corner cases related to allocation. While astonishing at first, these cases can be explained by the material presented during the class. We ask students to compare heap experiments with results obtained from allocations on the stack.

3.5 Session 5: Image processing pipeline

The last session consists in completing an image processing pipeline. It summarizes material from previous activities, but the implementation is done with the Windows API instead.

The goal is to complete the program to apply, in parallel, one or more effects to a set of images. Many effects are available such as sharpen, blur, and saturation. One thread is spawned for each pipeline stage and they are linked together with a blocking queue. The students are asked to implement thread management code and a blocking queue using semaphores. Restricting the queue length is important to limit memory usage in the application. Otherwise, the program could quickly load numerous images, thereby possibly exhausting available RAM.

Once the application is working properly, students are asked to propose a default queue length, and to justify it using their experimental results. We propose to the students to trace the number of images waiting in queues, according to the processing time of an image set. An example of such a run is shown in Figure 9. They can experiment with the effect of changing the maximum queue length on the total running time and memory consumption.



(a) Large heap allocation with fill

(b) Large heap allocation without fill

(c) Small heap allocation with fill

Figure 8: Heap allocation experiments

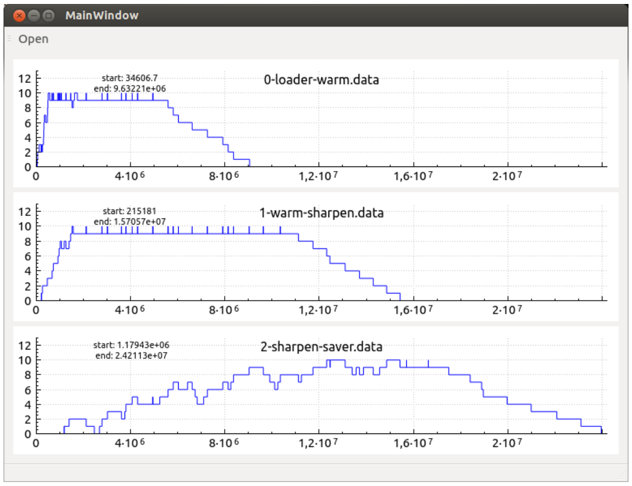


Figure 9: Queue size of the pipeline according to time

Moreover, the slowest stage in the pipeline can be identified when the input queue fills up. This activity has the benefit of exposing students to queueing theory in an intuitive manner.

4 Outcomes

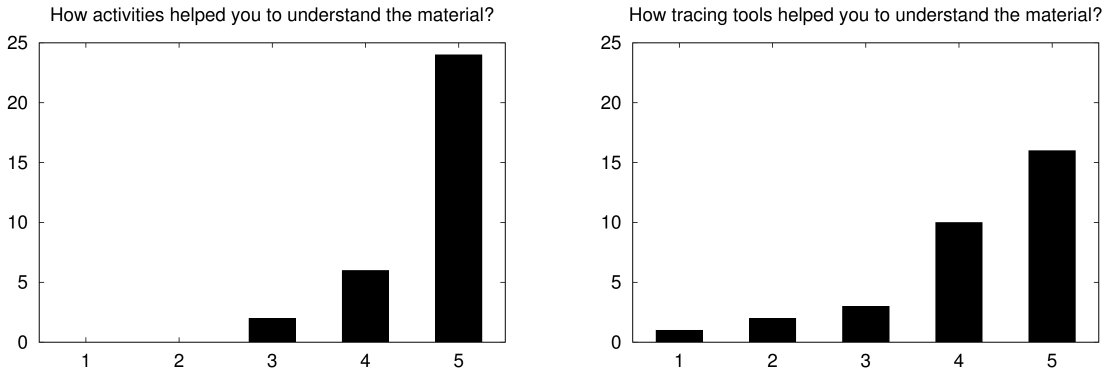
The above activities were used for teaching Operating Systems during the 2013 Fall semester. There were 58 students registered for the class.

Comparing the proposed activities to other approaches would require multiple groups with different material. However, it was not possible for logistic reasons. Instead, our goal is to verify that activities are well suited to the context at hand, to learn about related difficulties, and to document potential improvements. We invited students to participate in an online survey at mid-term and we conducted a focus group at the end of the semester. We plan to perform in the following semester a quantitative analysis on students grades to determine the effect of the new material.

4.1 Survey

A survey was performed after the third session. We asked students to which extent, on a five point scale, they considered activities helpful in understanding the course material. We also asked more specifically about the helpfulness of execution visualization tools. Summaries of answers are presented in figure 10. Students agreed to a large extent that activities were helpful. The rate is slightly lower for the question specifically about execution visualization tools.

We analyzed comments from students to understand the situation. We found that some students were overwhelmed by specific technical details related to usability of visualization



(a) Overall activities helpfulness

(b) Tracing tools helpfulness

Figure 10: Summary of surveys' answers

tools. These difficulties confirmed observations made during interactions with students. We present solutions to address this issue in the future directions section.

4.2 Focus group

Following the last session, we conducted a focus group to get feedback about the overall semester. Topics discussed covered material comprehension, resources quality, interest and motivation, difficulties encountered, and improvement ideas. The duration was set to one hour and half. An external person facilitated the meeting so that students would feel comfortable to speak freely. The objective was to compose a group of five to eight participants, and finally six volunteers were enrolled.

Many students said that visualization tools were an asset for understanding. Without the course, they would not have known about these tools. It was appreciated that the tools are free. Some of the participants installed the tools on their computer to work from home, but also mentioned that doing so was tedious.

One of the main issues raised by the group concerns the waiting time to get help. The ratio of 30 students for one laboratory assistant is too high. The type of activity proposed requires more resources to support students adequately, especially the first semester, before the material is revised based on the feedback.

In terms of content, it was suggested to trace another programming language, and to explore other system calls such as `mmap` and `mprotect`. Following these suggestions would incur relatively small changes. Other students expected to code in kernel mode. They suggested to add an activity that would consist in programming a device driver. Doing so would require a virtual machine setup in order to make it practical. The higher difficulty involved with kernel level programming, for instance testing and debugging, must be taken into account in the design of such activity; it may be more appropriate for advanced courses.

Opinions are divided about the amount of work involved. Some students took much more time to complete activities, while others would have liked to go further. The amount of work thus seemed to be an appropriate compromise. Additional help could reduce time lost, while optional activities could be proposed to enthusiastic students.

5 Future directions

We plan to improve the activities in several ways during upcoming semesters. We want to address usability issues with the trace viewer. The first issue concerns the location of the workspace directory. The workspace contains the trace index, and it must be on the local hard drive for optimal viewer performance. Unfortunately, since Eclipse is also used as IDE, the default directory is on the network drive of the students and this causes noticeable slowdown for trace navigation. In addition, two instances of Eclipse must be started, and it was one major source of confusion. We plan to use a repackaged subset of Eclipse for tracing purposes only. The workspace directory will be set automatically, and using a separate tool will avoid the confusion with the Eclipse IDE.

In order to simplify installation steps, we will focus on supporting Ubuntu. LTTng kernel modules are packaged for Ubuntu, and the system will compile modules on the fly for each kernel version installed. Currently, this must be done manually on Fedora. We will provide a pre-configured virtual machine to further simplify the setup. Once downloaded, the students will be able to run it and perform experiments from home.

In the short term, and for logistic reasons, we do not plan to add any activity involving kernel mode programming. However, we feel that this could be interesting in the future and we will continue to investigate its pedagogical value. For instance, an activity could consist in instrumenting a device driver to evaluate one aspect of its performance.

In terms of pedagogical approach, we want to evaluate the possibility of including more socio-constructivist aspects to these activities. We believed an online forum would be a good way to share knowledge, yet we observed low participation despite advertising it. We will instead evaluate the possibility to include a project presentation or programming competition.

6 Conclusion

We presented a way to integrate execution visualization into the Operating Systems course. We described five activities we designed, tools we used, and their purpose. Results of the qualitative evaluation shows that activities are helpful toward learning operating systems concepts. We identified future improvements to increase satisfaction. Finally, we discussed potential developments that could help continue improving these activities.

7 Acknowledgement

The authors thank Christian Giraldeau, Renaud Giraldeau and Joëlle Morrissette for their contribution to the pedagogical aspect of the project, to all students who provided valuable feedback, and to Philippe Doucet Beaupré and Dominic Gauthier for additional review.

References

[1] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating systems: design and implementation*, volume 13. Pearson Prentice Hall, 2006.

[2] Zuefu Zhou. Teaching an operating system course to cet/eet students. In *Proceedings of the 2009 American Society for Engineering Education Annual Conerence*, 2009.

[3] Michael D. Filsinger. Writing simulation programs as a tool for understanding internal computer processes. In *Proceedings of the 2004 American Society for Engineering Education Annual Conerence*, 2004.

[4] Steven F. Barrett, Daniel J. Pack, and Charles Straley. Real-time operating systems: A visual simulator. In *Proceedings of the 2004 American Society for Engineering Education Annual Conerence*, 2004.

[5] Luiz Paulo Maia, Francis Berenger Machado, and Ageu C. Pacheco, Jr. A constructivist framework for operating systems education: A pedagogic proposal using the sosim. *SIGCSE Bull.*, 37(3):218–222, June 2005.

[6] Wayne A. Christopher, Steven J. Procter, and Thomas E. Anderson. The nachos instructional operating system. In *Proceedings of the USENIX Winter 1993*, 1993.

[7] Charles L Anderson and Minh Nguyen. A survey of contemporary instructional operating systems for use in undergraduate courses. *Journal of Computing Sciences in Colleges*, 21(1):183–190, 2005.

[8] Timothy Bower. Using linux kernel modules for operating systems class projects. In *Proceedings of the 2006 American Society for Engineering Education Annual Conerence*, 2006.

[9] Mathieu Desnoyers and Michel Dagenais. Teaching real operating systems with the lttng kernel tracer. In *ASEE Annual Conference and Exposition, Conference Proceedings*, Pittsburg, PA, United states, 2008.

[10] Catherine Twomey Fosnot and Randall Stewart Perry. Constructivism: A psychological theory of learning. *Constructivism: Theory, perspectives, and practice*, page 8–33, 1996.

[11] Eclipse Foundation. Eclipse linux tools project. `http://www.eclipse.org/linuxtools/`, December 2013.

[12] LTTng team. Linux tracing toolkit next generation. `http://www.lttng.org/`, December 2013.

[13] Perf wiki. `https://perf.wiki.kernel.org/`, December 2013.

[14] Wine hq. `http://www.winehq.org/`, December 2013.