ABSTRACT
This paper is an experience report outlining a transformation of an operating systems course in a systems oriented master study program. We present the initial and current stages of the transformation and summarize the lessons learned throughout the process. To illustrate the effects of the transformation, we provide feedback gathered from the students who have completed the course during the transformation. We put emphasis on general software engineering experience, which may prove useful in balancing the many issues related to most courses with programming assignments.

Categories and Subject Descriptors
K.3.2 [Computers and Education]: Computer and Information Science Education; D.4.0 [Operating Systems]: General

General Terms
Human Factors

Keywords
teaching, operating systems, student assignments

1. INTRODUCTION
A proverbial advice attributed to Confucius says: I hear, I forget. I see, I remember. I do, I understand. The concept of understanding through doing is very relevant to software engineering education, which relies heavily on student assignments to convey the understanding of technical details that are typically difficult to present in lectures.

Such an arrangement requires proper timing of the assignments relative to the lectures, determining the appropriate depth of the assignments, and also involves significant effort spent on grading the assignments.

As far as assignments are concerned, software engineering education presents a particular challenge in that it often deals with issues only present in large projects. These issues are notoriously difficult to demonstrate in smaller assignments, which can lead to lectures acquiring the feel of beast fables – a lot of good advice is dealt out but never demonstrated in situations that the students experience first hand.

One solution to this issue is introducing specialized courses with large assignments, or requiring industrial internship, as a part of the study program. This approach, however, is limited by the size of the study program – an especially painful consideration in the two year master study programs frequently created off previous five year integrated study programs to comply with the Bologna Declaration [4].

We have therefore decided to put more emphasis on general software engineering issues even in specialized courses with programming assignments. This experience report describes such a change on the example of the operating systems course taught within Charles University.

The Charles University operating systems course has been designed to not only provide a wide overview of major system related concepts, such as process execution, process communication, memory management, memory virtualization and device handling, but to also foster sometimes less than perfect understanding of basic system related principles, such as the way in which machine code is executed by the processor or the way in which the processor communicates with devices.

We outline the course structure as it underwent a several years long transformation, adjusting various parameters of the assignments, and present the feedback gathered from the students during the transformation. The feedback concerns general software engineering experience and can help balance many issues related to the operating systems course in particular or similar courses with programming assignments in general.

In the following sections, we first outline the initial course structure, discuss our reasons for the transformation and detail its outcome. We proceed by outlining selected results of the several conducted student surveys. The summary of our own experience follows, with due conclusion.
2. INITIAL COURSE STRUCTURE

In the years prior to 2002, we have been using Nachos [6] from University of California at Berkeley as the instructional operating system, along with a subset of assignments corresponding to those originally published by the authors of Nachos.

Since writing even a minimum operating system from scratch is demanding, Nachos tries to avoid the danger of too difficult assignments by providing an operating system template in which the students are required to extend or replace the implementation of its individual parts. To shorten the lengthy and tedious compile-execute-debug cycle when developing an operating system on real hardware, Nachos runs the kernel of the operating system as a standard application on a host computer. Nachos then provides as a part of the kernel a simulated processor on which the user level applications are executed.

While using Nachos, our assignments were specified relatively vaguely, in terms of required functionality, and were graded on meeting these loose requirements. Since the code to demonstrate the functionality was written by the students themselves, and the code was reviewed mostly for potential design errors, a window for overlooking some bugs existed.

Software engineering concerns, such as team management or coding quality, were not an essential part of the grading since they were deemed outside the scope of the course.

Geared at making the assignments easy for the students, the architecture of Nachos introduces several features that are either rather unrealistic – such as the existence of dynamic heap during bootstrap, or result in additional complexity not found on real hardware – such as different encoding of integers in user and kernel memory.

Over the several years of using Nachos, we felt that this unrealistic setup, together with assignments that shielded students from low-level details by using high-level object-oriented interfaces, went contrary to our effort at making the assignments as close to practice as possible.

To deepen the understanding of the operating systems concepts and construction, as well as the general software engineering experience, we have transformed the course by modifying the assignments with the following goals:

- To create an environment that is more realistic than Nachos, but at the same time preserves some of the ease of debugging arising from running the operating system and user programs as a normal process on a host machine.
- To remove the soft cushions of abstraction arising from a well structured operating system template covering most of the basic functionality, into which students plug their enhancements without really having to think about many low-level issues.
- To put more emphasis on the software engineering aspects of the assignments, such as team management, coding practice and assignment documentation, which in turn should help the students to better manage complexity.

In 2001, we have started to work on an alternative infrastructure for the student assignments and phased out Nachos in the following year. Later, we have found out that we were not alone taking this decision [8, 11].

3. MODIFIED COURSE STRUCTURE

3.1 Environment Realism

To obtain a realistic environment while avoiding the major hassle with debugging, we have implemented a full system simulator based on the MIPS R4000 CPU [9]. Compared to a real hardware platform, the simulator is somewhat simplified – there is no firmware to interact with and the devices (a keyboard, a console, and a hard drive) are mapped directly to memory.

The simulator does not support the 64-bit mode of the R4000 CPU, floating point coprocessor, and memory caches. Other than that, the behavior of the R4000 CPU is preserved as much as possible to create a fairly realistic setup for the students to work with. This allows the students to use a stock GNU compiler toolchain1 to generate executable code for the simulated MIPS platform. The simulator also supports multiprocessor systems, which are becoming widely available.

Besides continuing the tradition of Nachos, the choice of the CPU was prompted by the fact that we wanted a real but simple CPU. This not only helps keep the simulator small, but also prevents students from getting lost in the peculiarities and complexity of CPU architectures such as IA-32.

3.2 Design Freedom

To avoid students plugging their work into an existing operating system template, we have modified the assignments so that students have to write a full kernel almost from scratch. To get around the need to code in assembly language, we provide an example code that bootstraps and supports threads with primitive memory management and crude synchronization. Using the assembly language fragments basically as they are, students are then required to implement synchronization primitives, complete thread support, virtual memory without swapping, and user mode processes.

From the software engineering perspective, this means that more emphasis is put on the design phase of the assignment. Using broad estimates for a project of about 5000 lines of code with no final deployment, the design phase can represent as much as 24 % of the overall development effort [3], which the students did not do previously.

The assignment requirements have evolved from vague functional requirements through a plain list of methods to be implemented to a detailed specification of the required in-kernel interface. We require the students to design their solutions while considering various non-functional trade-offs. In the assignment specification, we provide directions concerning standard usage patterns for different parts of the kernel to avoid trivial and unrealistic solutions. The quality of the solution is then included in the grading of the implementation.

From the software engineering perspective, this means that less emphasis is put on the requirements phase of the assignment, in exchange for the possibility of more rigorous testing of the in-kernel interface. Using the same estimates, the requirements phase can represent as much as 14 % of the overall development effort [3], which the students do not do now.

1GCC, GNU Assembler, GNU Linker, etc., typically built as a cross-compiler toolchain.
Given the level of realism we wanted to impose on the students, even simple assignment solutions with all the technical details to take care of present a significant challenge. We therefore give the assignments to teams of students rather than individual students. The usual size of the team is three students.

3.3 Grading

We have split our grading of the assignments into a separate grading of the implementation functionality, a separate grading of the coding quality, and a separate grading of the design documentation and presentation. The three parts are given equal weight. To further our goal of making the grading more consistent, we rely on having specific assignment requirements accompanied by functional tests. As the individual parts of the assignments are completed during the semester, each part has to pass an automated suite of functional tests, manual grading is only done at the end of the semester.

The tests exercise the student solutions and report inconsistencies in expected and obtained results. Even though the CPU simulation is completely deterministic, the tests are randomized to avoid static test cases. The randomization is driven by a pseudo random generator which can be seeded by a user defined constant, which makes it possible to reproduce all bugs discovered by the tests in a deterministic fashion, which is especially important when debugging race conditions. Since the tests are given to the students along with the assignment specification, they also help to make the requirements more precise.

4. EVALUATION

To evaluate the transformation, we use results from voluntary anonymous surveys among the students after completing the course, conducted in several semesters. We are aware that this is still a relatively small statistical sample taken under unstable conditions and would therefore caution against drawing any far-reaching conclusions, but we believe certain major trends can be observed, as follows.

The obvious question about the transformation from the initial to the modified course structure is how did it influence the perceived relevance and difficulty of the assignments. Perceiving the assignments as relevant and doable is a major factor in course impact [13].

In 2004, we have surveyed a mix of students who have passed the course in both its original and its modified structure (both groups were comparable in size). Figure 1 shows the perceived relevance of the assignments before and after the transformation. With the original structure, some 19% of the students did not see the contribution of the assignments to their understanding of operating systems as useful (marks 0 to 4). This unsatisfied group is gone with the modified structure of the course, where the overall average has also shifted slightly, from 6.45 to 7.83. This trend stays in the survey from early 2008, as seen in Figure 2, where the majority says the assignments have helped quite significantly. As expected from the changes outlined in the previous sections, the perceived overall difficulty of the assignments has increased after the transformation (see Figure 3). Most students consider the new assignments difficult given the size of the course, but surprisingly, practically no student suggested cutting down on the assignments (see Figure 4). Most advocate extending the credit rating of the course instead and some would keep the course as it is. Later surveys, asking on the scope of the entire lecture rather than just the assignments, also contain no majority suggestion for cutting down the work.

The introduction of tests, which are given to the students along with the assignment specification, was generally welcomed as extremely useful (see Figure 5). The role of tests has gradually shifted from a grading tool to a development support tool, in an interesting agreement with Extreme Programming [1]. We consider supporting this trend by creating more tests, ranging from simple functionality checks to serious stress test, and automated daily runs with on-line results.

An interesting aspect of the assignments was a spontaneous application of software engineering methods. Even though no software engineering technique was explicitly suggested, the 2008 survey shows students using and praising specific software engineering techniques – Pair Programming [15] and Surgical Team [5], code proofreading and unit testing.

A more thorough demonstration of software engineering issues is also indicated for team coordination and assignment debugging, which were reported in the early 2008 survey as two most significant obstacles just after general lack of time and understanding. We are still hesitant on the issue of debugging – we realize that our simulated environment is not nearly comparable with contemporary interactive debuggers, but on the other hand, we often witness a lack of debugging skills that can hardly be attributed to tools alone.
Figure 1: Before (left) and after (right) the transformation: Did the assignments help in your understanding of the principles of operating systems? (0 = not at all, 10 = irreplaceably)

Figure 3: Before (left) and after (right) the transformation: How does the amount of work compare with the credit ranking? (0 = too much work, 5 = adequate, 10 = too many credits)

Figure 4: Before (left) and after (right) the transformation: How would you alter the ratio between the amount of work and the credit ranking? (0 = less work, 5 = as is, 10 = more credits)
5. LESSONS LEARNED

During the transformation of the course, important issues have appeared related to finding an appropriate workload that can be imposed on the students, as well as the effect of the new arrangement on the workload imposed on the grading staff.

5.1 Assignments

Concerning assignments, we have initially tried to avoid giving too much work to individual teams by splitting the assignment into three parts corresponding to kernel subsystems, with each part serving as a foundation for the next part. The teams were then assigned one of the parts and – with the exception of teams responsible for the first part – implemented their assignment on top of previous part implemented by another team. This required assigning strict deadlines during the semester to each part and choosing only the best implementations for the later teams.

We have perceived this arrangement as very useful, since students acquired experience with modifying code written by someone else. As far as the specific work of individual teams goes, the perceived disadvantage of teams working on later assignments was assumed to be balanced by the fact that they had more time to get acquainted with the development environment, whereas the teams working on the first assignment had to start basically from scratch, in an unknown environment, and had a very strict deadline on finishing their assignment.

Unfortunately, even though our functional tests exercised the assignments quite thoroughly, they could not ensure entirely bug-free code and in several cases, students working on the later assignments were bit by hidden bugs in code they have not written. Moreover, we have assumed that students would study and understand the code of their predecessors and extend the code base with the implementation of their own assignment. This assumption turned out to be wrong. Students were in fact reluctant to study foreign code and instead tried to touch it as little as possible and treat it as a black box. In part, this was fueled by the fact that students were given code that passed the tests and thus tended to consider it bug-free.

Facing these issues, we have come to a conclusion that while this approach is certainly good for getting experience with extending foreign code, it is not entirely fair and our expectations of student behavior were somewhat unrealistic. We now require all teams to implement essential subset of all the kernel parts, and an extended version of one kernel part of their choice.

The deadlines, originally introduced to enforce a timely delivery of the code needed by other teams, were left in place to encourage the teams to start working as soon as possible. Since the assignments can be deceptively simple, we have found it very important to make sure that the students realize the real complexity of the work early on. Still, the majority of the work on the assignments is done just a few days before the particular deadline (as seen in Figure 6), which leads to a strong pressure to postpone the deadlines. Although the inequality of the workload during the time period between the particular deadlines might have many reasonable explanations, such as more focus on designing at the beginning and more focus on coding at the end of the period, other clues suggest that the real reason is the lack of proper time management, discipline and a manifestation of planning fallacy (tendency to systematically underestimate task-completion times).

As a final observation on the assignments, the introduction of the functional tests served to verify student progress without increasing the workload on the grading staff. However, we have also noted that some teams started to optimize their work to pass the tests, rather than focusing on the more general requirements.

5.2 Grading

As far as the grading goes, the new arrangement resulted in an increased workload on the grading staff. The existence of tests gives more confidence in the consistency of the gradings of the implementation aspect of the assignment, but the rest of the grading related to quality of the implementation and coding practices has to be done manually by reviewing code and design documentation.

Besides grading the assignments, we also wanted to give the students useful feedback on their work, including both the architectural decisions and the coding quality. Initially, this was limited to a few sentences to point out flaws that caught our eye, but gradually, it has evolved into a detailed list that pointed out what is wrong and why and what would be a better way to do it.

We have noted that many of our comments were related to coding quality. This confirmed our belief that grading non-functional aspects of student assignments is important. The only drawback is effort that needs to be spent on reviewing student code to provide useful feedback. On average, one person can spend about six hours reviewing the work of a single team.

This could be reduced if factors such as code quality were not graded too rigorously. While we would indeed rather focus at grading the operating-systems-specific aspects, we believe that code quality is as important as the other aspects of grading. Without prior coding experience, students often have very little or no notion concerning good coding practices, which is why we believe that students should be given some instruction on this aspect of software development. However, we are also aware of the difficulty of including a full sized course purely on code quality, as explained for example in [10], in the already tightly packed curricula.

5.3 Topics

Perhaps not quite a surprise, adjusting the assignments to provide a more thorough demonstration of software engineering issues caused the students to encounter more problems related to those issues. Since students tend to expect receiving instruction on any potential source of problems, this in turn creates pressure to include the software engineering issues among the topics presented during lectures, somewhat diluting the focus on operating systems. We hope this problem can be solved by better synchronization between the operating system course and general software engineering courses.

6. RELATED WORK

Using hardware simulators in operating system courses has become quite common and there are several simulators designed specifically for educational purposes. They differ mainly in the simulated processor architecture and the degree of realism [7]. Even though we have developed our own system simulator, it was mainly because we could not find
one to suit our needs at the time we needed it. Since the goal of this paper is sharing experience from student feedback, we do not provide any comparison of simulator features.

A common practice in teaching is to use an existing operating system and give the students assignments to modify the system. The operating system used for teaching is typically a special purpose operating system that runs in a simulator, such as Nachos [6] or OS/161 [8], or a general purpose operating system that runs on real hardware, such as Minix [14] or Pintos [11]. Again, given the purpose of this paper, we want to avoid comparing different general purpose and educational operating systems. We will, however, point out a few similarities and differences between their approach and ours.

The OS/161 along with its MIPS-based hardware platform called System/161 [8] is particularly interesting because its development and deployment in teaching courses occurred at roughly the same time. In case of OS/161, the main reasons seem to have been the deficiencies of Nachos, whereas in our case, the abstraction of the assignments from the hardware was a strong factor.

The authors of OS/161 chose to replace Nachos with another, more realistic implementation, but kept the concept of assignments based on extending a fully featured operating system framework. The framework is quite large, 19000 commented lines of code in kernel space with 7000 additional lines in user space. Since this amount of code is hard to understand in short time, students are directed to specific parts of the framework and required to read code relevant to the particular assignment before starting their work.

In contrast to OS/161, we only give students a minimalistic example of a system that has trivial preemptive round-robin scheduling of a fixed number of kernel threads and has a simple first-fit kernel memory allocator, roughly 3000 lines of commented code in all. As a result, our assignments cover fewer operating systems concepts – no file systems, no complex device drivers, no networking – but the concepts we do cover have to be implemented basically from scratch, requiring the students to perform architectural decisions. This is consistent with the fact that our course is aimed at more advanced, master students, instead of undergraduate students.

Simple bootstrap framework and incremental assignments also allow us to avoid problems mentioned by the authors of OS/161, such as having to create certain subsystems in a primitive or outright faulty way in order to leave some work for the students. In general, though, we seem to share the same experience concerning undetected bugs carried over from earlier assignments, resulting in difficult-to-debug problems later.

Finally, the authors also comment on poor state of the MIPS development toolchain, which made them consider different processor architecture. While we have also experienced a few problems of this kind, neither of them were critical. Since the MIPS core seems to have gained popularity in embedded devices, the support of the MIPS platform in the GNU compiler toolchain is being improved lately and there are known workarounds for the remaining issues.  

The Pintos instructional operating system [11] also originated as a Nachos replacement for an undergraduate course. Unlike System/161, it runs on a standard simulator such as Bochs [2] or QEMU [12], or on real hardware, and its supported processor architecture is IA-32. While there are certain advantages in learning such a widely used architecture, we believe that a simpler RISC architecture such as MIPS is better for understanding the basic principles without dealing with complicated architecture-specific details. Note that the device support for such an architecture would be also more difficult, but the Pintos course assignments do not seem to require implementing that, except for changes in the existing timer device driver.

Similarly to Nachos and OS/161, Pintos also comes as a semi-complete framework, with four smaller assignments that extend thread support, implement syscalls, add virtual memory swap support, and improve the filesystem implementation. A reference solution of the assignments totals 3000 lines of modifications. This means that the same difference from our approach applies as in the case of OS/161.

The grading system of Pintos assignments seems to be almost identical to ours – tests are provided both for development support and grading, but the solutions have to be general and not just pass the tests. Good design documentation and source code quality is also required and graded.

The assignment documentation provided to students not only covers the requirements, but also gives development
suggestions such as how to divide work among team members or the order of general steps needed to complete the assignment. A statistics of the reference solution in terms of number of changed lines in each file is also presented for the workload estimate.

7. CONCLUSION

We have described an operating systems course transformation, with the goals of deepening the understanding and making the students perceive the course assignments as more relevant and useful. While this came at the cost of the assignments being seen as more difficult, our survey feedback suggests that the difficulty is seen not as a negative aspect that needs to be removed, but as a necessary factor that should be preserved. This leads us to believe that it is worth keeping the current course structure in the future, addressing some of the remaining problems.

From a broader perspective, we would conclude that there are drawbacks to using simplified programming assignments, which are not related to the operating systems education or the software engineering education in particular. Even when the assignments are carefully tailored to simplify the nonessential aspects of the problem, as was the case in our course, the result can be perceived as more artificial and less useful for understanding.

Even with the waning position of operating systems among the things a software engineer is expected to understand, our experience can be generalized. In the times of shortening the curricula in an attempt to cater to fast track education programs, it is worth knowing that unsimplified assignments serve better to foster understanding and that students seem to very much appreciate this.

In the education context, this would advocate the need for assignments that exercise multiple aspects of a discipline, even if they are more difficult and require more effort than a scope of a single course would warrant.

7.1 Acknowledgments

This work was partially supported by the Grant Agency of the Czech Republic project GD201/05/H014 and by the Ministry of Education of the Czech Republic (grant MSM-0021620838).

8. REFERENCES