Experiences in Teaching an Educational User-Level Operating Systems Implementation Project

Adam J. Aviv, Vin Mannino, Thanat Owlarn, Seth Shannin, Kevin Xu, and Boon Thau Loo
{aviv, vinm, towlarn, sshannin, kevinxu, boonloo}@cis.upenn.edu
University of Pennsylvania

ABSTRACT

The importance of a comprehensive implementation component for undergraduate Operating Systems (OS) courses cannot be understated. Students not only develop deep insight and understanding of OS fundamentals, but they also learn key software engineering skills that only a large development project, such as implementing an OS, can teach. There are clear benefits to traditional OS projects where students program or alter real (Linux) kernel source or extend educational OS implementations; however, in our experience, bootstrapping such a project is a huge undertaking that may not be accessible in many classrooms. In this paper, we describe a different approach to the OS implementation assignment: A user-level Operating System simulation based on UNIX preemptive signaling and threading constructs called `ucontext`. We believe that this variation of the implementation assignment provides many of the same educational benefits as traditional low-level projects without many of the expensive start-up costs. This project has been taught for a number of years at the University of Pennsylvania and was recently overhauled for the Fall 2011 semester. This paper describes the current version of the project and our experiences teaching it to a class of 54 students.

Keywords

Education, User-Level, Undergraduate, Operating Systems, Instruction Experience

1. INTRODUCTION

The undergraduate Operating System (OS) course is a standard litmus test for Computer Science (CS) majors throughout the world. At the core of nearly all OS courses is a large scale implementation assignment, or sequence of such assignments, that explore key OS features. These projects are crucial: students learn how to design and build large systems; often working in groups, students learn the benefits (and pitfalls) of programming as a team; and finally, students learn the intricacies of modern OSes, and as a side-effect, how to really use and understand them.

However, designing and supporting a large implementation project is an immense undertaking. We surveyed OS projects at the top CS programs in the country[1], and found that the vast majority of the implementation projects involve developing low-level kernel code[2]; that is, they require students to develop low-level code that can boot on physical or virtual hardware or in a simulated environment. Many of these course have long histories, developed over many years by a dedicated faculty member and accompanying teaching staff. They often require additional support, such as specialized software or simulators (*e.g.*, Nachos [5], Xinu [6], Simics [2], Pintos [8], GeekOS [7]) and/or dedicated machines and labs.

The start-up cost of such course projects is immense, and the rewards can be equally great. At the University of Pennsylvania, we have developed a different strategy to the OS implementation project that reduces much of the overhead. With the same educational goals as low-level implementation assignments, we have developed a strict user-level OS simulation project, named *PennOS*, that can be developed on any commodity UNIX-based OS installation without requiring students to have super user access, specialized hardware, or additional software.

At the core of the *PennOS* project is an often overlooked standard UNIX library, the *user context* library, that enables a single process to share resources amongst different execution points, much like threading. The user context library only requires user level access and is standard on UNIX based OSes, and it is what enables *PennOS* to reduce much of the overhead without sacrificing the educational benefits. In this paper, we describe our experience teaching *PennOS* in an undergraduate OS course at the University of Pennsylvania in the Fall 2011 semester. We designed *PennOS* to balance the following three goals:

- **Exposure to wide range of OS concepts.** First, to complement lecture material on OS concepts, students gain "hands-on" experience in actually building an OS that incorporates a significant portion of the concepts learned in class. Based on preemptive signaling and user contexts (UNIX's `ucontext` library), students are exposed to a wide range of OS concepts – most importantly kernel-/user-level separation, process life-cycle, priority scheduling, signaling, file-systems, and shell development.
- **Ease of configuration.** We require that *PennOS* run on commodity PCs with no changes to the underlying operating system or need for installing/running virtual machines or use of specialized software. This makes it easy for system administrators to support the course even with large class sizes, or when there are multiple project courses within the same department. As a result, the *PennOS* project is easily extended or adapted to fit the needs of the class without affecting the requisite lab support.
- **Software engineering.** Third, *PennOS* requires students to program in groups of four, working together to develop a complex system with approximately 6K lines of code on average. To get started, students have to read through substantial documentation regarding the `ucontext` library and develop their own test code for learning. Students build their

[1] As ranked by the US News and World Reports [9].

[2] Similar survey findings were reported here [3] using a larger sample of 98 schools: 50% use kernel-level implementations.

Week	Material	Project
1	Introduction	Project 0
2	Process/threads, system calls	
3	Concurrency and synchronization	Project 1
4	Scheduling	
5	Deadlocks	
6	Midterm and memory management	
7	*PennOS* project discussion	Project 2
8	Virtual memory	
9	Virtual memory	
10	Disks and file systems	
11	File systems	Milestone
12	Remote procedure calls	
13	Network programming	
14	Distributed file systems	
15	Wrapup and midterm	Demo

Table 1: Course Schedule

OS from scratch, and for many students, this constitutes the most substantial implementation project in their undergraduate curriculum. As a result, it is important to encourage and teach strong software engineering concepts. This includes requiring students to make use of code repositories for collaborative development, and to develop components separately with future integration in mind.

The first two goals are often opposing forces in course projects. A hyper realistic development environment that exposes students to the gory details of developing an OS from scratch requires specialized hardware and resources. We argue that *PennOS*'s approach provides a good balance between these two goals. While students may not be exposed to the mechanics (and pain) of kernel-level programming, the `ucontext` library still provides a high degree of realism, in exposing students to low-level development challenges.

Finally, it is important to note that we *do not* argue that *PennOS* is a "drop-in" replacement for traditional, low-level implementation assignments; rather, our intention is to present an alternative assignment that covers many of the same key concepts, is easy to manage, and hence, can be deployed and made more accessible in resource-constrained environments and large classrooms alike.

Detailed project descriptions and lecture slides are available for viewing at [1]. Further materials, including example code and project demonstration plans are available to instructors upon request. We welcome feedback from the community, and look forward to further developments of *PennOS*, with your support and participation.

2. COURSE OVERVIEW

To put *PennOS* in context, we first provide an overview of the OS course taught at our university. Our course is offered annually in the Fall semester, and it is a required course for CS majors in the School of Engineering Applied Sciences and an elective for CS majors in the School of Arts and Science. Most students take the class during their junior or senior year, and the average class size for the past three years is 44. In the last offering, enrollment reached 54, and this in part reflects the increasing number of CS majors.

Historically, the OS course had an accompanying half-credit lab section, but this separate lab was recently removed from the curriculum. Consequently, the course required restructuring so that the core OS implementation concepts previously covered in the lab could be integrated into a single full credit course with many more students participating in the project. As a result, we overhauled the project sequence to better fit the new demands of the class. Table 1 presents the course schedule and topics, as well as the project assignment timeline (based on release date).

Below we describe the sequence of projects, all of which are to be programmed in C. In total, the projects are worth 55% of the final grade. The remaining 45% consists of two midterms, written homework assignments, and class participation. Refer to [1] for more details on the course, including detailed project documents.

- **Project 0: *Timing Shell*.** Working individually, students program a simple shell that times process execution using `alarm()`, and will kill the running process if it executes longer than a threshold. This project is worth 5% of the overall grade.

- **Project 1: *Feature Shell*.** Working in pairs, students program a feature-rich shell with pipes, redirection, foreground and background processes, job control, and asynchronous signal handling. This project is worth 15% of the overall grade.

- **Project 2: *PennOS*.** Working in groups of four, students program an OS simulator with a preemptive priority scheduler and a file system mount; additionally, *PennOS* boots into a shell similar to *Feature Shell*, which links the scheduler portion and the file system implementations. This project is worth 35% of the overall grade.

The goal of the project sequence is to incrementally dive into the design requirements of an OS by investigating the UNIX system call interface. In the *Timing Shell* and *Feature Shell*, students are exposed to the functionality and behavior of the core system calls (*e.g.*, `fork()`, `wait()`, `read()`, `write()`), and in *PennOS*, students must implement many of those system calls within their *PennOS* simulator. Although students cannot always port code from one project to the next, each clearly builds upon the previous; however, an inability to complete one part does not render the following assignments intractable.

Students have over six weeks to complete the *PennOS* project, which is released after the first midterm. Halfway through *PennOS*, each group presents their progress in a milestone meeting, in which the teaching staff provide feedback on the group's development progress (and can also steer errant groups back onto track). The project culminates in group demonstration, during which the teaching staff performs a mini code review, tests functionality, and orally quiz groups on their development. Most of the grading is completed during the demo, and we were able to finalize and report grades within a week of submission.

In the rest of this paper, we will primarily focus on describing the *PennOS* project and the user context library that enables it.

3. PENNOS

PennOS is composed of three parts: process management and scheduling; file system and I/O; and shell integration. This division allows students to develop the priority scheduler and the file system independently (typically in groups of two). Once completed, students glue them together during shell development, which requires accessing the entire *PennOS* system call API.

Students often reported that the kernel development portion of the project was more challenging than the file system development. A desirable (and somewhat unexpected) outcome of the varying difficulty between these two components is that groups self-organized their development teams based on programming strengths. It was not uncommon for students with stronger programming backgrounds to work on the kernel development, while less experienced programmers worked on the file system portion. This division of labor ensured that all group members contribute to the *PennOS* development before the final integration. Consequently, we observed that students with strong *PennOS* implementations were members of groups where everyone contributed in some way to the final system.

Scheduling in *PennOS* is event driven, based on clock ticks occurring every 10 milliseconds. On a clock tick, a `SIGALRM` is delivered by the host OS and handled by invoking the *PennOS* scheduler, which chooses the next process to run. Processes in *PennOS* are represented by user contexts, which are part of the built-in UNIX library and enable programs to share their resources across different execution paths.

The file system in *PennOS* uses a File Allocation Table (FAT) and is contained within a single file on the host OS. It is mounted in *PennOS* in a loopback-like fashion. One of the challenges students face is how to appropriately write-through to the underlying file and how to manage file descriptors. It is important to note that the development of the core file system API (*e.g.*, reading, writing, creating and deleting files) can occur completely independently of the scheduler. We emphasize this by requiring students to develop separate programs from *PennOS* that can read and write to the file system by linking to the *PennOS* file system libraries.

The last part of the project involves developing a shell for *PennOS* to boot into. The shell's features touch all major parts of the kernel and file system implementation, and the integration between the two disjoint developments was reported as one of the most challenging parts. We required the shell to support the following features: foreground and background processes with job control; synchronous child waiting; a complement of built-in functions, *e.g.*, `cat`, `nice`, `sleep`, *etc.*; and redirection with truncation and append modes.

In the rest of this section, we present more details on the requirements for *PennOS*. We pay particular attention to the user context library and how we presented it to students. The user context library is fundamental to the project and was completely unfamiliar to students prior to the course. We also focus more on the challenges of the projects, as reported by the students, rather than on the low-level details. For more information about the particulars of *PennOS*, please refer to [1].

3.1 User Contexts

A user context, or `ucontext`, describes an execution stack of a running function, and it allows for the current state of execution to be saved (*i.e.*, preempted) and restarted later. In *PennOS*, students represent a process as a `ucontext`, scheduling them in and out as appropriate. Additionally, they can program a special `ucontext` to represent the kernel (and `init` for extra credit).

Programming directly with `ucontext` could be considered arcane – even the most proficient UNIX programmers may not have used them – but `ucontext` are implicit to all programs, especially those that rely on any form of preemption like signal handling and threading. Learning the `ucontext` library is also beneficial to students and aids their general understanding of the mechanics behind process/thread management. User contexts are what enable *PennOS*'s user-level development, and in this subsection, we focus on how we present and teach user contexts to students.

```
typedef struct ucontext {
  struct ucontext *uc_link;
  sigset_t        uc_sigmask;
  stack_t         uc_stack;
  ...
} ucontext_t;
```

Figure 1: The `ucontext_t` Structure Type-Definition

Introducing user contexts. When presenting the `ucontext` library to students, we found that examples from signal handling are most accessible because they relate well to previous projects and lectures. As a starting point, we refer students to the signal handling mechanisms they used in the *Feature Shell* project. In that project, students were expected to used the `sigaction()` system call for asynchronous signal handling. First, we direct students to `sigaction()`'s UNIX manual and its description of the `sigaction` structure that defines the signal handler, particularly to the advanced type definition for the handler function, `sa_sigaction`,

```
void (*sa_sigaction)(int, siginfo_t *, void *);
```

and its accompanying description:

```
This function receives the signal number as its
first argument, a pointer to a siginfo_t as its
second argument and a pointer to a ucontext_t
(cast to void *) as its third argument.
```

That is, the signal handling function is wrapped within a `ucontext` that is different from the `ucontext` of the main execution path of the program. This is natural: handling a signal requires the current execution to be preempted to run some other function (the signal handler), after which execution is returned to the original execution point. That is precisely what `ucontext` are designed to enable.

Once students grasp the key functionality of `ucontext` (as a mechanism for a process to maintain different execution states), it is an easy leap for them to represent a `ucontext` as a *PennOS* process. In other words, a `ucontext` is an abstract structure and mechanism for *PennOS* to share its resources amongst different execution states. Likewise, a process is an abstract structure and mechanism for an OS to share the CPU resources amongst different execution states.

Mechanics of user contexts. The next educational challenge is teaching students the mechanics and requisite system calls to explicitly start, save, and swap in and out a `ucontext`. The `ucontext` library primary structure is a `ucontext_t`. To initialize a `ucontext_t`, the `getcontext()` system call is used. This initializes the structure with appropriate (often machine dependent) values; however, many of the other fields of the `ucontext_t` still need to be set.

The three primary structural values are presented in Figure 1. The first value, `uc_link`, refers to the `ucontext` that should be set once this one completes. The `uc_sigmask` and `uc_stack` are the signal mask and the stack to use when running this `ucontext`, respectively.

Notice that the function that is to run when the `ucontext` is started is not specified in the structure. This is because the `ucontext_t` is agnostic to the code that will execute when it is set, and in fact, the same `ucontext_t` (in memory) can be used repeatably with different functions. To actually assign the

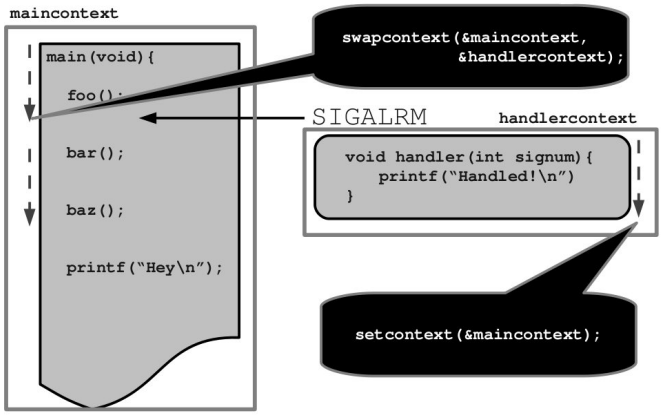

Figure 2: Swapping and Setting Contexts as Exemplified by Signal Handling

function to run, the `ucontext_t` must be made with a call to `makecontext()`, and it can be remade repeatably and reused for a different function execution. However, if this step is neglected, as students quickly found out, the program will often `SEGFAULT` when setting the `ucontext`. This was one of the most common errors, as well as overflowing the stack allocation.

The last two requisite library functions are used to set and swap user contexts. Their type definitions are as follows:

```
int setcontext(const ucontext_t *ucp)
int swapcontext(ucontext_t *oucp, ucontext_t *ucp)
```

The distinction between these two functions is a point of confusion for many students, even though both essentially perform the same task of setting the current `ucontext` to `ucp`. The only difference is that `swapcontext()` additionally saves (and makes) the current `ucontext` in `oucp` while `setcontext()` only sets the context without saving.

To illustrate the difference between these two functions, we again used a signal handling example. Referring to Figure 2, when the `SIGALRM` is delivered, first the current `ucontext` must be saved and the handler `ucontext` swapped in. This is accomplished with a call to `swapcontext()`. Once the handling function returns, control is returned to the main `ucontext` with a call to `setcontext()` since the handler context does not need to be retained.

Finally, once students are familiar with the mechanics of the `ucontext` library, the first exercise is to alter a "Hello World" program. Figure 3 presents the sample "Hello World" program we provided students in the project write-up, and we encouraged students to complete the following exercises: (1) Use `uc_link` to have execution return to main; (2) Do the same without using `uc_link`; and (3), Program an infinite loop of "Hello World" prints using the same `ucontext_t` structure.

3.2 Kernel Development

In this section, we describe the high-level requirements of the kernel, which includes programming a simple priority scheduler and managing the process life-cycle. At this point, after understanding `ucontext`, we encouraged groups to split into two teams of two. One team continues with kernel development, and the other team starts developing the file system; however, we encouraged teams to inform the other throughout the development because they need to ensure that the parts can correctly interface later.

The first task of the kernel development team is to outline the primary structures and process-centric system call API for their *PennOS*. In particular, they need to define the Process Control Block (PCB) structure. Just as in traditional OSes, the PCB stores all requisite information about a running process, such as its process ID, parent process ID, open file descriptors, list of children process, state flags, *etc.*. The next two tasks involves implementing the scheduler and handling process state and signaling.

```
void f() {
  printf("Hello World\n");
}

int main(int argc, char * argv[]) {
  ucontext_t uc;
  void * stack;

  //initialize ucontext_t structure
  getcontext(&uc);

  //set its values
  stack = malloc(STACKSIZE);
  uc.uc_stack.ss_sp = stack;
  uc.uc_stack.ss_size = STACKSIZE;
  uc.uc_stack.ss_flags = 0;
  sigemptyset(&(uc.uc_sigmask));
  uc.uc_link = NULL;

  //assign the function f to execute at start
  makecontext(&uc, f, 0);

  //set the context to uc
  setcontext(&uc);

  //setcontext() doesn't return on success
  perror("setcontext");

  return 0;
}
```

Figure 3: Hello World using `ucontext`

Priority scheduling. We required students to program a very simple priority scheduler with three priority queues: -1, 0, and 1. The lower the priority number the more frequently the process should be scheduled, *e.g.* the shell should run with priority -1 because it is interactive. We required that processes with priority -1 get scheduled 1.5 times as often as those with priority 0, which get scheduled 1.5 times as often as processes with priority 1. Within each queue, processes are to be selected in a round-robin fashion.

Granted, this style of priority scheduling is non-traditional, but since I/O blocking is still controlled by the host OS, it is difficult to implement a classic multi-level scheduler. However, an approximation of I/O blocking can be achieved by having a special I/O `ucontext` (scheduled like any other process) which queues and handles I/O requests bound for the file system. *PennOS* processes with an enqueued request are considered blocked until all reading/writing is completed. This enables students to implement the elevator algorithm, and we allowed students to attempt this as extra credit, which was successfully completed by two teams.

Process states and signals. Another requirement of the kernel is tracking processes' life-cycle states. *PennOS* maintains many of the same process states as traditional OSes; however, there is only a single explicit blocking situation, when a process is blocking on a `wait()` call. The process life-cycle from spawning, running, and termination is as expected, and once a process terminates, it becomes a zombie and is placed in the zombie queue of its parent so that it may be reaped. We did not require an `init` process, and as such, the procedures for handling orphans was to immediately remove them and clean up their resources. We note, however,

that implementing an `init` process (to run at priority 1) is an easy extension and was an optional extra credit which four teams completed successfully.

We also require that *PennOS* handle stopping, continuing, and killing processes, which requires a form of simple signaling. Instead of using signal queues and handler functions, we allowed students to treat signals as prompts to the kernel to take a particular action. For example, if the `SIGKILL` signal is directed at a target process, the kernel marks that process as terminated and moves it to the zombie queue of its parent. Similarly, if a `SIGSTOP` or `SIGCONT` signal is delivered, the target process transitions from a runnable to stopped state, or *vice versa* respectively.

We felt that implementing a signal queue and handler functions may be too much of a burden for students, and as such, we do not require asynchronous signal handling. This inhibits a `SIGCHLD`-like signal, which also simplifies the shell's child waiting procedure. It should be noted that asynchronous signaling and handling is a reasonable extension to *PennOS*, particularly given the nature of the `ucontext` library, and in future revisions, we intend to make it an extra credit option.

3.3 File System

The file system development team is tasked with implementing a File Allocation Table (FAT) based file system with 1MB file block. The file system data exists within a single file on the hosts OS and is mounted in a loopback-like manner. We only require the file system to have a single directory, but we allowed groups to implement multi-level directories as extra credit (which was successfully completed by four groups).

To facilitate testing and debugging, we require students to provide two programs that can read and write to the file system from the host OS. These two programs link to the *PennOS* file system library and invoke all the key file system system calls. This allows the file system team to test and debug their implementation without interfering with the kernel development. It also encourages students to carefully plan their API so that the code can be ported into the main kernel at integration time.

The details of the file system are rather straightforward, and students easily handled the data layout required. The most challenging part for students was maintaining consistency with the file system on disc, reading and writing it into memory and ensuring that all edits get written back to disc. We suggested that students use `mmap()` with a write-through mapping for the file blocks that need altering, and many groups did, but often students found it easiest just to map the entire file system into memory. We limited the file system size, based on the size of the FAT (just 512 entries), and so this turned out to be a reasonable choice.

Another area where students reported challenges was in maintaining file descriptor consistency. This is particularly important because once the file system is integrated with *PennOS*, there are really two types of file descriptors: descriptors for files open from the file system and descriptors for standard input and output. We required that *PennOS* provide a *single* interface for reading and writing, just `read()` and `write()` system calls. The I/O module must therefore be able to handle a descriptor appropriately depending on the intention of the user. Students discovered that some carefully placed conditionals cleared up their confusion.

3.4 The Shell and Integration

Once each part is near completion, groups merged their developments by implementing a basic shell with a number of built-ins. To illustrate, Figure 4 shows sample code written by students for the primary shell loop. From this sample code, one can see that

```
//giant switch on PennOS built programs
void *func = switch_on_builtins(args[0]):

//fork new process/ucontext to execute func
int new_pid = p_spawn(func, 0, arg_count, args);

//handle redirection, if any
if (p_handle_redirection(new_pid,
                         arg_count,func_args) < 0) {
  p_kill(new_pid,S_SIGTERM); //something bad happened
  continue; //reprompt for input
}

//add to jobs queue for job management
update_pid_queue(&jobs_queue_head, new_pid, 1);

//fore_or_back set earlier in code
if (fore_or_back == FOREGROUND) {
  //set to foreground
  curr_foreground_pid = new_pid;
} else {
  // run in background ...
}

//wait on processes that changed state, e.g., finished
struct pid_info wait_result = p_wait(NOHANG);

//keep waiting if the process that changed state was
//not the foreground process
while (fore_or_back == FOREGROUND &&
       wait_result.pid != curr_foreground_pid) {
  wait_result = p_wait(NORMAL);
}
```

Figure 4: Student sample code: The primary shell loop for `fork exec wait` equivalents in *PennOS*. Note that all p_* functions refer to `PennOS` system calls.

many of the underlying *PennOS* system calls are required to write the shell. In addition, we also require the following shell features:

- **Redirection:** The shell must provide `stdin` and `stdout` redirection to and from the file system in both append (>>) and truncate (>) mode.
- **Job control:** The shell must provide simple job control (*e.g.*, `jobs`, `bg`, and `fg`) and the ability to start programs in the foreground and background.
- **Synchronous child waiting:** The shell must wait on its children in a consistent manner, and without a `SIGCHLD` signal, this can be done synchronously, *e.g.*, using a `W_NOHANG` like flag in `wait()`.
- **Built-in and library functions:** The shell must also provide a number of built-in and library functions to facilitate use and testing. For example, the standard programs like `cat`, `ps`, and `sleep` were required, but we also required groups to have a `busy` and an `orphan` program which loops indefinitely or purposely creates an orphan process, respectively. (See Figure 5 for student sample code of the `cat` program, named `s_cat` for "shell cat".)

Each of these requirements is dependent on completing the kernel and file system components; for example, redirection depends on the file system integration, and job control and child waiting depends on the kernel. Students aptly noted that when testing their shells, a number of assumptions and bugs were exposed, as was the intention. One of the project goals is to teach software engineering lessons, and interfacing and accessing other code is a major part of that intellectual development.

```
void s_cat(char *func_args[]) {
  char user_str[1024];
  int input_len;

  if (func_args == NULL) {
    p_exit();
    return;
  }

  if(func_args[1] != NULL) {
    //redirect the stdout
    if (p_redirect(curr_pcb->pid, F_READ,
                   func_args[1]) < 0) {
      p_exit();
      return;
    }
  }

  while (1) {
    //only operate in foreground
    if(curr_foreground_pid != curr_pcb->pid) {
       p_kill(curr_pcb->pid,S_SIGSTOP);
    }

    input_len = p_read(user_str);

    if (input_len > 0) {
      user_str[input_len-1] = '\0';
      p_write(user_str); //writes to FlatFAT fd or stdout
      p_write("\n");
    } else {
      p_exit();
      return;
    }
  }

  p_exit();
  return;
}
```

Figure 5: Student sample code: The builtin program `cat` in *PennOS*. Note that p_* functions refer to *PennOS* system calls.

3.5 Logging and Testing

An important requirement of *PennOS* is logging. This not only aids the teaching staff, but also guides the students in debugging. We required students to use a consistent format for easy parsing and grading. The logging events and the format are presented in Figure 6.

```
[ticks] SCHEDULE PID QUEUE PROCESS_NAME
[ticks] CREATE PID NICE PROCESS_NAME
[ticks] SIGNALED PID NICE PROCESS_NAME
[ticks] EXITED PID NICE PROCESS_NAME
[ticks] ZOMBIE PID NICE PROCESS_NAME
[ticks] ORPHAN PID NICE PROCESS_NAME
[ticks] WAITED PID NICE PROCESS_NAME
[ticks] NICE PID OLD_NICE NEW_NICE PROCESS_NAME
[ticks] BLOCKED PID NICE PROCESS_NAME
[ticks] UNBLOCKED PID NICE PROCESS_NAME
[ticks] STOPPED PID NICE PROCESS_NAME
[ticks] CONTINUED PID NICE PROCESS_NAME
```

Figure 6: Logging Events.

From these events, all the major state changes and events are present, and when helping students debug their projects in office hours (and in the milestone and demo), this was an invaluable resource. One strategy we found useful was decreasing the clock tick rate (*e.g.*, to 500 milliseconds) and examining the log as it is written using `tail`.

Another test we found useful for debugging the scheduling behavior is to monitor the CPU usage of *PennOS* using a tool like `top`. For example, if there are two programs running in *PennOS*, the shell with priority -1 and a busy wait program with priority 0, the expectation is that *PennOS* will use approximately 40% of the host OS CPU resources (when no other computation-intensive programs are running). This is because the shell is blocking (from the host OS) on a read from `stdin` which is scheduled 1.5 as frequently as the busy program which is consuming CPU resources.

When testing the file system, the most valuable tool is `hexdump`. This allowed students to visually inspect the FAT and file block layout. The two additional programs which students are required to create also greatly aid in debugging because they focus on the core reading and writing of the file system. These programs can also be easily run alongside a standard debugging tool, such as `gdb` and `valgrind`. Unfortunately, this is not easily done with the full *PennOS* development because `gdb` and `valgrind` do not gracefully handle the `ucontext` library. However, students were still able to work through most of their problems using the methods described above, and it is important to note that in more low-level development these tools are also cumbersome.

4. EVALUATION AND EXPERIENCE

Student feedback on *PennOS* has been overwhelmingly positive. The course ratings and enrollment are the highest in the past three years (since the lab component was removed). In the course evaluation, students found the project "very rewarding", "very interesting and intellectually challenging", and felt the projects made them "more comfortable around low-level C than ever before." They also reported that the project "was an incredible experience" and "hoped to have more classes like this one." 62% of students gave the class the highest possible score of 4 (on a scale from 0 to 4) on the amount learned in this class in terms of knowledge, concepts, skills and thinking ability.

At the same time, the class was also rated the second most challenging course in the department for both graduate and undergraduate students, and was the most challenging undergraduate course in the Fall 2011. 57% of students gave the class the highest difficulty rating of 4. Students found the course "grueling," particularly in the second half of the semester. However, students also pointed out that despite the difficulties and time spent, they "learned more in this course compared to any of my other cs courses", and "highly recommended the course for every cs major."

We did receive valid criticism in the reviews. Students responded that the lectures and projects were not well integrated. While the topics (e.g. scheduling, file systems) covered in class are conceptually featured in the projects, students at times felt that there were in fact two classes running side-by-side, one involving textbook material and one involving the *PennOS* project sequence. This is in part a result of merging the OS class syllabus with the now defunct lab syllabus, and further, this was the first time *PennOS* was taught without a lab section. With the benefit of hindsight, an important next step in the OS class development is to revisit the lectures and better integrate them with the project so that key concepts are reinforced in and out of the classroom.

5. RELATED WORK

There is a rich history of OSes designed specifically for educational purposes in the literature; most notably, Xinu [6] and Nachos [5]. Additional educational OSes such as PintOS [8] and GeekOS [7] are also used at universities and colleges throughout the world, and with the advent of mobile technology, OS implementation projects based on developing networking principals have

also been proposed, such as PortOS [4].

There are two major difference between *PennOS* and these proposals. First, *PennOS* is specifically designed to run at the user-level and take full advantage of the UNIX system call interface. It is not designed to run on real or simulated hardware, but the implementation experience and educational lessons are an excellent primer for more advanced developments. Second, *PennOS* is a standalone project, built by students from scratch without the need of additional software. The proposals listed above are designed for students to extend and build upon an existing code base. While this is an important software engineering lesson, the freedom and design experience of building comprehensive software from scratch resulted in students feeling strong ownership for their work. We were surprised at the many different ways student chose to implement their *PennOS*.

6. CONCLUSION

In this paper, we describe our experiences at introducing OS concepts through building *PennOS*, a user-level operating system simulation. The project has been very well received by students, receiving positive feedback and high teaching evaluations.

PennOS balances realism and ease of management while all implementation occurs at the user-level, only requiring a commodity UNIX environment without any additional software. Central to its success is the `ucontext` library, a UNIX built-in library that enables processes to share resources amongst different execution points. The use of the `ucontext` library ensures that students are exposed to many of the same concepts as they would be in low-level projects, and since *PennOS* is a user-level development, it eases the configuration demands and reduces the learning curve such that students can complete a large-scale OS simulator within a relatively short time period.

Moving forward, we plan to tighten the integration of the project and the lecture material, as mentioned in Section 4. One key piece of OS design lacking from the project is memory management, and in future revisions, we plan to incorporate an MMU implementation in the project (it was an extra credit in the current version). Other features like networking and socket layers are also possible project extensions.

7. ACKNOWLEDGMENTS

The *PennOS* project presented herein is based on previous versions and has had many contributors over the years. Particularly, we would like to acknowledge contributions by Sandy Clark, Micah Sherr, Eric Cronin, Stefan Miltchev, Guarav Shah, Hee Hwank Kwak, Stuart Eichert, Scott Raven, Jon Kaplan, Robert Spire, Dianna Xu, and Insup Lee. Additionally, we would like to thank Di Mu, Cam Nguyen, Angela Wu, and Henry Jin You for allowing us to use their sample code.

References

[1] CIS 380: Operating Systems. `http://www.cis.upenn.edu/~cis380/`.

[2] Simics. `https://www.imics.net`.

[3] Charles L. Anderson and Minh Nguyen. A survey of contemporary instructional operating systems for use in undergraduate courses. *J. Comput. Small Coll.*, 21:183–190, October 2005.

[4] Benjamin Atkin and Emin Gün Sirer. Portos: an educational operating system for the post-pc environment. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, SIGCSE '02, pages 116–120, 2002.

[5] Wayne A. Christopher, Steven J. Procter, and Thomas E. Anderson. The Nachos Instructional Operating System. In *USENIX Winter 1993 Conf.*, 1993. `http://www.cs.washington.edu/homes/tom/nachos/`.

[6] Douglas E. Comer. *Xinu Operating System*. Prentice Hall, 1987. `http://www.cs.purdue.edu/research/xinu.html`.

[7] David Hovemeyer, Jeffrey K. Hollingsworth, and Bobby Bhattacharjee. Running on the bare metal with geekos. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, SIGCSE '04, pages 315–319, 2004.

[8] Ben Pfaff, Anthony Romano, and Godmar Back. The pintos instructional operating system kernel. In *Proceedings of the 40th ACM technical symposium on Computer science education*, SIGCSE '09, pages 453–457, 2009.

[9] U.S. News University Directory. Top Computer Science Schools & Best Ranked Colleges, 2010. `http://www.usnewsuniversitydirectory.com/graduate-schools/sciences/computer-science.aspx`.