Awk-Linux: A Lightweight Operating Systems Courseware

Yung-Pin Cheng and Janet Mei-Chuen Lin

***Abstract*—Most well-known instructional operating systems are complex, particularly if their companion software is taken into account. It takes considerable time and effort to craft these systems, and their complexity may introduce maintenance and evolution problems. In this paper, a courseware called Awk-Linux is proposed. Awk-Linux can be crafted relatively more easily and it does not depend on any hardware simulator or platform. The basic hardware functions provided by Awk-Linux include timer interrupt and page-fault interrupt, which are simulated through program instrumentation over user programs. Course projects based on Awk-Linux provide source code extracted and simplified from a Linux kernel. Results of this study indicate that the projects helped students better to understand inner workings of operating systems.**

***Index Terms*—Computer science education, courseware, operating system kernels, operating systems.**

I. Background

A number of possibilities exist when an instructor wants to design hands-on programming assignments for an undergraduate operating systems (OS) class. An assignment may range from simple tryouts of system calls/Windows APIs, to a multiphase implementation of a complete operating system with a minimal set of services. The latter approach is impractical, if not entirely impossible, because a complete operating system is far too large and complex. The system-call/API approach, however, fails to provide students with better insight into the inner workings of an operating system. Instructional operating systems strike a middle ground between the above two approaches. With an instructional OS, students may gain experience with the hidden "insides" of an operating system by developing or enhancing the key features of an OS without working from scratch.

Although some researchers (e.g., [2] and [3]) have argued that the use of instructional operating systems is detrimental because they are removed from reality to a certain extent, no other alternatives have been shown to provide better OS learning experience than the use of instructional OS courseware.

Several instructional operating systems have been developed in past decades. MINIX [4], XINU [5], Nachos [6], and OSP [7] are some of the best known. The systems developed more recently include TOS [8], PortOS [9], Topsy [10], OS/161 [11], GeekOS [12], KayaOS [13], and BabyOS [14]. A survey of contemporary instructional operating systems used on 98 campuses, as well as the types of assignments designed in association with those systems, can be found in [15].

Systems developed earlier, such as MINIX [4] and XINU [5], were designed to run directly on bare machines. They can be viewed as "total" instructional operating systems. However, since many of their kernel features are greatly simplified or scaled down, they are not complete general-purpose operating systems. This type of instructional operating system also presents porting problems, since porting a system from one platform to another is no easy task.

More recent instructional operating systems, such as Nachos [6] and OSP [7], mostly rely on hardware simulators, which can be executed under host operating systems such as Linux or Windows. The kernel and the hardware simulator typically run as user processes. The advantage of this approach is that the machines are not limited to educational use only. Moreover, since the systems do not depend on real hardware, the problem of having to catch up with hardware advances is considerably reduced.

For an instructional operating system to be operative, typically a minimal set of services has to be implemented. Nachos, for example, contains 4000 to 5000 lines of code for its minimal set of services. Systems that rely on bare machines, such as XINU and 5–MINIX, require even more code. For example, MINIX had about 30 000 lines of source code in the kernel alone. As such, these systems are by no means "simple."

The issue is even more complicated if their companion software is taken into account. For example, a hardware simulator must be acquired first. After that, a C compiler, which compiles C programs to the instructions of the simulated hardware, is needed. A debugger may also be needed if the systems are to be friendlier to its users. Although these companion programs may be free and available from third parties, porting them to a different platform or another hardware simulator can be a tremendous task since part of the kernel code of these companion programs inevitably depends on the hardware features provided by the hardware simulator.

In this paper, an instructional OS courseware called Awk-Linux is presented. Section II begins with a short description of Awk-Linux and then presents two basic tools which simulate two fundamental hardware functions. The course projects based on Awk-Linux are described in Section III, which is followed by an evaluation of the course project in Section IV and

Manuscript received August 5, 2006; revised September 18, 2007.

Y.-P. Cheng is with the Department of Computer and Information Engineering, National Taiwan Normal University, Taipei, Taiwan, R.O.C. (e-mail: ypc@csie.ntnu.edu.tw).

J. M.-C. Lin is with the Graduate Institute of Information and Computer Education, National Taiwan Normal University, Taipei, Taiwan, R.O.C. (e-mail: mjlin@ntnu.edu.tw).

Digital Object Identifier 10.1109/TE.2007.912571

the conclusion in Section V. This paper is an expanded version of a preliminary version published in [16].

II. Awk-Linux

Awk [17], a utility under Unix and a programming language for performing text-processing tasks, is useful for translating files from one format to another, creating small databases, adding additional functions to text editors like vi, and so on [18]. When used as a text-handling tool, Awk allows users to define string patterns to be matched, where string patterns are specified using regular expressions. The input file is parsed line by line. When a match occurs between a string and one of the patterns, a block of Awk statements is executed, either replacing the string or rewriting the whole line. Awk is used in this study as a program instrumentation tool to simulate basic hardware functions.

A. Program Instrumentation

Program instrumentation, a well-known technique in the field of software engineering, is often used to monitor dynamic execution of programs by inserting extra statements or instructions into these programs. In most cases, the inserted statements do not alter the behavior of the monitored programs.

In Awk-Linux, selected C user programs are instrumented and combined into an executable program named awkos.exe, including necessary kernel features extracted from the real Linux kernel. When students run awkos.exe, they can observe user programs being executed concurrently in a time-sharing manner. Context switches are simulated by jumping from one user program to another, as will be described below.

Program instrumentation typically requires the help of language processing tools. Awk was selected for this project. A C language parser could also do the job, although working on the parser of a popular programming language seems an overkill solution. Other viable alternatives include such well-known Unix tools as lex and sed. Lex is not as powerful as Awk because Awk provides plenty of language constructs for manipulating strings (records) in a line. Moreover, neither lex nor sed views an input file as lines of records, making them less appropriate for use in this study. A complicated sed script also suffers from low readability.

B. Simulation of Time-Sharing and Timer Interrupts

In a real operating system, concurrency is realized via hardware timer interrupts and context switches, which create a time-sharing execution environment for user programs. In a conventional instructional operating system, a hardware simulator has the responsibility of creating the illusion of concurrency.

The following example illustrates how a timer interrupt is simulated in Awk-Linux. Two user programs—test1.c and test2.c (shown in Fig. 1)—are merged and converted into an executable main program (shown in Fig. 2) to create the concurrency illusion. Note that some parts of the C code in Fig. 2 are omitted to conserve space.

The instrumented source code of Fig. 2 is generated mechanically by invoking the following command:

```
gawk makekernel.awk test1.c test2.c
```

```
------------test1.c---------------
 #include<stdio.h>
 main()
 {
    int i;
    for (i=0; i<200; i++)
    {
       printf("This is    111111%d\n", i);
       sys_sleep(10);
    }
 }
```

```
------------test2.c---------------
 #include<stdio.h>
 main()
 {
    int i;
    for (i=0; i<200; i++)
    {
       printf(">>>>>> 222222%d\n", i);
       sys_sleep(50);
    }
 }
```

Fig. 1. Two user programs before program instrumentation.

```
#define stepincr(lableno,syscall) \
        stepctr++; \                                // increase the step counter
by 1
        if (syscall == 0) {\
             if (stepctr >= qtime) {\               // if step counter exceeds a
threshold,
                      stepctr = 0; \                 // we prepare to emulate
triggering a clock tick.
                      prev = current; \
                      current->addr = lableno;   \   // save the return address
for returning from interrupt
                      do_timer(); \                  // call do_timer() to emulate
a clock interrupt
                 } \
            } \
            If (current != prev) {\                  // if pointer current points to
another process
                                                     // control block (PCB), we
should prepare to
                                                     // jump to the code of the
new user process.
                   prev = current;
                   switch(current->addr)\
                        case0: gotoL0; case1: gotoL1; case2: gotoL2; case3:
gotoL3; case4: gotoL4;
                        case5: gotoL5; case6: gotoL6; case7: gotoL7; case8:
gotoL8; case9: gotoL9;
                        case10: gotoL10; case11: gotoL11; case12: gotoL12;
case13: gotoL13;
                        case14: gotoL14; \
                 } \
            } \
  //definition of prog2
  Int USER2_i;
  //definition of prog3}
  Int USER3_i;
  main() {
  //extrainitialcode..........................}
  //code of test1.c
  stepincr(4,0); L4:
  stepincr(5,0); L5:
  stepincr(6,0); L6: for(USER2_i=0;USER2_i<200;USER2_i++) {
  stepincr(7,0); L7: printf("This is 1111\%d\n",USER2_i);
  stepincr(8,0); L8: saveaddr(9);sys\_sleep(10);}
  stepincr(9,1); L9: }
  sys_exit();}
  //code of test2.c
  stepincr(10,0); L10:
  stepincr(11,0); L11:for(USER3_i=0;USER3_i<200;USER3_i++) {
  stepincr(12,0); L12:printf(">>>>>>2222 \%d\n",USER3_i);
  stepincr(13,0); L13:saveaddr(14);sys_sleep(50);
  stepincr(14,1); L14:}
  sys_exit();}
  stepincr(15,0);
}
```

Fig. 2. test1.c and test2.c are merged mechanically by the program instrumentation script.

where gawk is the GNU Awk. Makekernel.awk is the Awk script which combines the programs provided, test1.c and test2.c in this case, into a single program and inserts extra code to simulate concurrency. The combined file (called awklinuxos.c) will be linked with other kernel functions and become an executable program called awkos.exe. When awkos.exe is executed, students will see that test1.c and test2.c produce output in an interleaved fashion as if context switches actually occur. The program terminates when execution of both loops in test1.c and test2.c comes to an end. Note that all local or global variables in a user program are renamed by adding a prefix "USERi" to

```
void do_timer()
{
    / *decrease one tick*/
    struct   task_struct*p;
    p = current;
    if (p->counter <= 0)
    {
        p->counter = 0;
        p->need_resched = 1;
    }
  If (p->need_resched)
        schedule();
}
```

Fig. 3. The interrupt handler do_timer() under Awk-Linux.

each. By making all variables global, the combined main program would not confuse variables from different user programs.

The trick for creating the illusion of concurrency is achieved with the mechanically generated goto statements. Each statement in a user program is instrumented by adding a macro stepincr() and a goto label to the statement (see lines 29–43 in Fig. 2). The body of macro stepincr(), which varies depending on the given user programs, is described in the top portion of Fig. 2 (lines 1–21). The stepctr() macro in front of each statement allows the user to count how many statements a user program has executed. When the counter stepctr in the macro exceeds a threshold, the macro calls subroutine do_timer(), which is the timer interrupt handler in the Linux kernel, to simulate a tick of the timer. Once a user program exhausts its time quantum, do_timer() calls schedule() to select a process from the ready queue to run next. The pointer of the PCB (process control block) of the current process is updated accordingly. When the update occurs, stepincr() retrieves the last address (a goto label) of the newly scheduled process and jumps to that statement.

The source code of do_timer() under Awk-Linux is listed in Fig. 3. Those who are familiar with the Linux kernel will easily discern that the source code is a simplified version of the real Linux kernel, including its data structures, and that the Linux kernel coding style is preserved.

C. Simulation of Paging Hardware

A more complicated instrumentation in Awk-Linux is to simulate Intel Pentium's two-level paging architecture. To explain the instrumentation, a user program test3.c is given in Fig. 4 and the instrumented source code is shown in Fig. 5. The main idea is to intercept each variable reference and simulate the address translation that is typically done by a two-level paging hardware. For example, the statement "$i = 0$" of the nested *for* loop in Fig. 4 is substituted by the statement "*memint("i", 16384,1) = 0." Function memint() returns an integer pointer pointing to another address where variable i is actually allocated. Parameter 16384 in the above statement is the logical address of variable i computed by the Awk script. Since variable i is declared after the declaration of an integer array p, which begins with logical address 0, variable i gets the logical address of 16384.

```
-------------test3.c------------------
main()
{
     int p[4096];
     int i, j, k;
     for (i=0; i<=3; i++)
     {
          printf("valuei=%d\n", i);
          {
               For (j=0; j<10; j++)
               {
                    k=i*1024 + j;
                    p[k] = j;
               }
          }
     }
     For (i=0; i<10; i++)
     {
          For (j=0; j<=3; j++)
          {
               k=j*1024 + i;
               printf("TEST1=%d\n",p[k]);
          }
     }
}
```

Fig. 4. test3.c: A user program created to test the paging memory management.

```
stepincr(4,0);L4:
stepincr(5,0);L5:
stepincr(6,0);L6:     for((*memint("i",16384,1))=0;(*memint("i",16384,1))<=3;
      (*memint("i",16384,0))++){
stepincr(7,0);L7:        printf("valuei=%d\n",(*memint("i",16384,0)));
stepincr(8,0);L8:
      for((*memint("j",16388,1))=0;(*memint("j",16388,0))<10;
      (*memint("j",16388,0))++){
stepincr(9,0);L9:
      (*memint("k",16392,1))=(*memint("i",16384,0))*1024
+ (*memint("j",16388,0));
stepincr(10,0);L10:               (*aryint("p",0,(*memint("k",16392,0)),1))=
      (*memint("j",16388,0));
stepincr(11,0);L11:     }
stepincr(12,0);L12:   }
stepincr(13,0);L13:   for((*memint("i",16384,1))=0;(*memint("i",16384,0))<10;
      {(*memint("i",16384,0))++){
stepincr(14,0);L14:
      for((*memint("j",16388,1))=0;(*memint("j",16388,1))<=3;
      (*memint("j",16388,0))++) {
stepincr(15,0);L15:
      (*memint("k",16392,1))=(*memint("j",16388,0))*
      1024+(*memint("i",16384,0));
stepincr(16,0);L16:               printf("TEST1=%d\n",
(*aryint("p",0,(*memint("k",16392,0)),0)));
stepincr(17,0);L17:        }
stepincr(18,0);L18:   }
sys_exit();
}
```

Fig. 5. test3.c after instrumentation and translation.

As Fig. 5 shows, each variable reference in test3.c is replaced by either of the following C functions, where TYPE can be int, char, or float (see the bottom of the page). Function aryTYPE() is used to translate array variables. Parameter rwflag is set by

TYPE *memTYPE (char *var_name, unsigned int logical_addr, int rwflag);
or
TYPE *aryTYPE (char *var_name, unsigned int logical_addr, int index, int rwflag);

the Awk scripts if the value of a variable is modified. This information will be used to set the dirty (modified) bit when the paging hardware is simulated.

A basic kernel function alloc_one_frame() is provided by Awk-Linux to allocate one frame and return a starting address in the form of 0xhhhhh000. Function memint() is responsible for mapping logical address 16384 onto the address in this frame. In other words, this study implements a two-level paging scheme to simulate Intel x86, which divides the logical address into 10,10, and 12 bits and accesses the page tables twice to obtain another 32 bit address. When page tables are not properly set, the page-fault interrupt handler do_page_fault() is called to simulate the phenomenon of triggering a page-fault interrupt.

D. *Characteristics of Awk-Linux*

The Awk scripts described above are useful in designing any project which aims at enhancing students' understanding of such topics as process scheduling, semaphores, virtual memory management, etc. To design a new course project, the easiest way is to modify existing projects, as described below. The first author has extracted and simplified many Linux kernel functions that can be used by an instructor to design new projects. However, if an instructor wants to devise a new project that differs considerably from the existing ones, he or she will need to extract and simplify Linux kernel code on his/her own.

It is also worth noting that, with Awk-Linux, writing new Awk scripts is necessary only when new hardware functions have to be simulated. Therefore, neither instructors nor students need to worry about any details in Awk scripts or the merged execution file awkos.c.

Compared with similar systems such as Nachos, OSP, and XINU, Awk-Linux also exhibits the following characteristics.

a) Awk-Linux can be run partially, without the support of a kernel with minimal set of services. That is, OS components can be implemented on an as-needed basis when designing course projects for students. For example, in the sys_sleep() project to be described below, the provided source code has only 222 lines of code, including comments.
b) There is no dependence of Awk-Linux on any particular hardware simulator or platform. Only Awk and GNU C compiler are needed. As such, maintenance and porting are much easier.
c) The time and effort needed to craft Awk-Linux are greatly reduced for the reasons mentioned in a) and b).
d) The Linux kernel coding style is preserved since the provided source code is extracted and simplified from the real Linux kernel.
e) Awk-Linux runs in deterministic execution mode. Real operating systems run non-deterministically. Nondeterministic programs are hard to test and debug, which can be too much of a burden for students.
f) Awk-Linux treats each C statement as atomic, hence it is impossible to use Awk-Linux to design a project that would allow students to experience race conditions at the CPU instruction level. However, race conditions can occur at a higher level, such as when two processes access a shared linked list. Awk-Linux can be precisely tuned to trigger race conditions at this level.

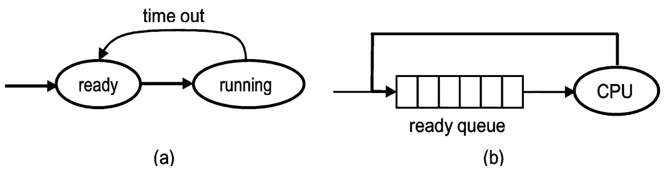

Fig. 6. A two-state scheduler.

III. Designing Projects Around Awk-Linux

Awk-Linux was used in the Operating Systems course taught by the first author to classes of junior students between 2003 and 2005. The textbook adopted was *Operating System Concepts* by Silberschatz [19], and the main topics taught in the semester-long course included process, multithreading, scheduling, process synchronization (including deadlocks), virtual memory management, file systems, and I/O. Among them, the topics of process, scheduling, process synchronization, and virtual memory management were covered by Awk-Linux. The Sections III-A–III-D describe how programming projects were devised for these topics.

A. *Scheduler: Implementation of Sys_sleep()*

The purpose of this project was to provide students with hands-on experience in manipulating process control blocks, process states, and process queues, by requiring students to convert a two-state scheduler into a three-state scheduler. The provided source code for this project included two user programs, test1.c and test2.c, and necessary kernel functions.

Code for test1.c and test2.c has already been shown in Fig. 1. Fig. 3 displays some of the kernel functions, which include a timer interrupt handler do_timer() and a simple round robin scheduler. The kernel provided to students was a two-state scheduler as shown in Fig. 6. The body of the system call sys_sleep() was left empty. Students were asked to fill in details of sys_sleep() so that the output of test1.c and test2.c would interleave at a ratio between 1:3 and 1:5. Some of the C macros in the Linux kernel were also provided for students to use.

To complete the assignment, students first needed to know that process control blocks in the Linux kernel were maintained as a doubly-linked list. They then had to modify the timer interrupt handler, add new fields and flags to the data structure of the process control block, create a waiting queue, and finally manipulate the linked list when a sleep process was waken up. They also needed to modify the kernel into a three-state scheduler, shown in Fig. 7. Once the implementation was completed, students could write their own programs to verify their results.

B. *Semaphores: Implementation of Semget() and Semop()*

semget() and semop() are semaphore system calls provided by Linux. Instrumentation for this project was the same as that for the previous project. Two user programs containing the calls to semget() and semop() were provided. Students were required to implement the system calls so that the two user programs

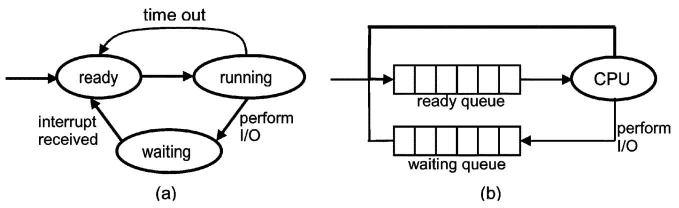

Fig. 7. A three-state scheduler.

would synchronize correctly. Specifically, students must perform the following activities:

- set up the data structure of a semaphore that included a wait queue;
- move a process to the wait queue of a semaphore when a semaphore call should be blocked;
- wake up the blocked process of a semaphore when a signal was invoked.

C. Virtual Memory: Implementation of Do_Page_Fault()

This project required students to complete the page-fault interrupt handler (do_page_fault()) so they would have a deeper understanding of paging hardware and have hands-on experience in page table management. Students were provided with necessary functions and macros to allocate a frame and manipulate page table entries. The paging hardware was the two-level paging scheme described in the previous section. When user programs were executed and page faults occurred, students would notice that do_page_fault() was called and the logical address which had caused the page fault was passed as a parameter. The body of do_page_fault() was left empty. Students had to complete do_page_fault() correctly to have user programs run properly. They needed to allocate frames for page tables, set up frame addresses in the page table entries, and set the valid bits. An incorrect implementation would result in a core dump.

D. Virtual Memory: Implementation of Shmget() and Shmat())

[AU: In heading, two closing parentheses correct?--ed.]Two user programs were provided for this project. Students needed to complete shmget() and shmat() so that the user programs could share a memory area. To implement the two system calls, students had to understand the memory layout of a user program. The process control block required more complicated data structures, such as vm_area_struct in the Linux kernel, to manage different virtual memory areas in a user program. Once the shared pages were allocated, it was necessary for students to establish page table entries correctly so that a page could be shared by two user processes. This project was originally intended to help students improve their understanding of the concepts of interprocess communication, shared memory, process image management, manipulation of page tables, and attaching a shared memory region to a pointer, but was later discarded as being too difficult for an undergraduate course.

IV. Evaluation and Discussion

A questionnaire survey and a focus-group interview were conducted at the end of the semester to collect students' feedback. This section reports these findings.

A. The Questionnaire Survey

As mentioned above, three of the four projects (that is, except Project D) described in the previous section were used in the course. They are indexed as 2, 4, and 5 in Table I. (Projects 1 and 3 did not involve the use of Awk-Linux, hence they were not included in the survey.)

Based on a five-point Likert scale, the figures displayed in Table I are the numbers of students whose answers ranged from *strongly agree* (5.0) to *strongly disagree* (1.0). The rightmost column presents the average of students' responses.

For Project 2 (Scheduler), it can be seen from Table I that out of a total of 46 students, 43 students (93%) either *strongly agreed* or *agreed* that the project helped them better understand the related concepts. The corresponding figures for Project 4 (Semaphores) and Project 5 (Virtual Memory) are 39 (85%) and 41 (89%), respectively. As to whether the projects helped them gain a better view of how the concepts were realized in a real operating system, 40 students (87%) *strongly agreed* or *agreed* for Project 2 and 38 students (83%) for Project 4. The result was not as impressive for Project 5, showing only 26 students (57%) who agreed. The students gave all three projects an average of 3.5 as to how difficult they felt each project was, indicating that the difficulty level of these projects seems appropriate.

B. The Focus Group Interview

Eight students participated voluntarily in the interview. In terms of their semester grades, six of them were above average in class, two were among the struggling students. In the following is a summary of opinions expressed by students in the interview.

1) Regarding the Tracing of Linux Kernel Code: Almost all students said that they struggled hard to understand the provided source code because of the many macros and pointer operations contained in it and also because they had little experience tracing code written by others. As one above-average student put it:

> "*It took me a lot of time. The names of the macros looked so similar that it was hard to tell one function call from another. I would comment out some of the statements if I really could not figure out what was going on. It was real tough at the time of the first project. But after I managed to finish the first one, I got much more comfortable with the later ones.*"

Comments like this did not come as a surprise. In fact, it was exactly the instructor's intention to train students in tracing code as well as in acquainting themselves with the Linux coding style. Students need to understand that kernel programmers, in everyday practice, opt to apply programming tricks to increase kernel performance. Readability has never been a primary concern to them. Although students may get frustrated in the beginning by the daunting macros and pointer manipulation, they are simultaneously realizing the important idea that performance is a top priority in kernel programming. Such an experience would be useful to them if they one day need to "get their hands dirty" with kernel programming.

Most students did appreciate the purpose of such training. As one student stated:

TABLE I
SURVEY RESULTS

Question A: Did the project help you understand the related concepts?
Question B: Did the project help you understand how the concepts were actually
implemented in a real operating system?
Question C: Did you find the project easy or difficult?

Project 2: Scheduler: Implementation of *sys_sleep()*

Question	strongly agree	agree	neutral	disagree	strongly disagree	*average*
A	9	34	3	0	0	*4.1*
B	3	37	4	2	0	*3.9*
	very difficult	difficult	neutral	easy	very easy	*average*
C	3	18	25	0	0	*3.5*

Project 4: Semaphore: Implementation of *semget() and semop()*

Question	strongly agree	agree	neutral	disagree	strongly disagree	*average*
A	3	36	4	3	0	*3.8*
B	3	35	7	1	0	*3.9*
	very difficult	difficult	neutral	easy	very easy	*average*
C	4	16	24	2	0	*3.5*

Project 5: Virtual Memory: Implementation of *do_page_fault()*

Question	strongly agree	agree	neutral	disagree	strongly disagree	*average*
A	3	38	4	1	0	*3.9*
B	2	24	15	5	0	*3.5*
	very difficult	difficult	neutral	easy	very easy	*average*
C	2	18	26	0	0	*3.5*

"*It's important to be able to trace code written by others. I learned a lot from it. The training helps to develop one's skill in 'exploring' the unfamiliar code. Some code was really amazing. I would never have written something like that myself.*"

Though most students admitted that the handout provided with each project was well written, including the description of the macros used in the project, they still suffered from the difficulties inherent in the obscure Linux kernel code. In light of these comments, instructors who are considering devising similar projects may want to provide an upfront tutorial at the beginning of the semester to minimize students' frustration.

2) Regarding the Benefits of Awk-Linux Projects: All students agreed that the Awk-Linux projects deepened their understanding of the operating systems concepts. As one student put it:

"*We were given two weeks for each project. I spent much time reading the related chapters of the textbook first; otherwise I would not know how to proceed with the project. After having a clear idea of the related concepts and the requirements of the project, I found it rather easy to finish it.*"

3) Regarding Which Other Topics Should be Covered by Awk-Linux Projects: The present study implemented four projects on three OS topics, namely scheduling, semaphore, and virtual memory. When asked what other topics might be good candidates for the instructor to design similar projects, one student proposed "file system," because he felt that a solid understanding of file-system internals was crucial to the implementation of I/O-efficient code that involved heavy file accessing.

V. CONCLUSION

A novel alternative to constructing instructional operating systems was presented in this paper. Awk-Linux does not depend on any hardware simulator or platform, and allows OS components to be implemented on an as-needed basis when designing course projects for students. As such, Awk-Linux can be crafted relatively more easily. Moreover, the Linux kernel coding style is preserved in Awk-Linux since the provided source code is extracted and simplified from the real Linux kernel. Results of this study indicate that the projects designed around Awk-Linux helped students better to understand the inner workings of operating systems, especially the Linux Kernel.

The first author of this is maintaining a website which contains the projects described in this paper. Interested readers may contact him via e-mail for the Web address. Currently, more projects under Awk-Linux are being designed to cover other operating system topics. In addition to the topics typically covered in an OS textbook, new and enhanced techniques related to non-distributed operating systems, as suggested in [20], are also probable topics for future projects surrounding Awk-Linux.

REFERENCES

[1] T. D. Wagner and E. K. Ressler, "A practical approach to reinforcing concepts in introductory operating systems," in *Proc. 28th ACM Technical Symp. Computer Science Education (SIGCSE)*, 1997, pp. 44–47.
[2] A. Perez-Davilla, "O.S. bridge between academia and reality," *ACM SIGCSE Bull.*, vol. 27, no. 1, pp. 146–148, Mar. 1995.
[3] ***[AU: Please provide name of publisher--ed.]***D. Jones and A. A. Newman, *A Constructivist-Based Tool for Operating Systems Education*. Denver, CO: , 2002.
[4] A. Tanenbaum and A. Woodhull, *Operating Systems: Design and Implementation*, 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 2006.

[5] D. Comer and T. Fossum, *Operating System Design: The Xinu Approach*. Englewood Cliffs, NJ: Prentice Hall, 1984.

[6] W. A. Christopher, S. Proctor, and T. Anderson, "The Nachos instructional operating systems," in *Proc. Winter USENIX Conf.*, 1993, pp. 479–488.

[7] M. Kifer and S. A. Smolka, *OSP: An Environment for Operating System Projects*. Reading, MA: Addison-Wesley, 1991.

[8] T. Nicholas and J. A. Barchanski, "TOS—An educational distributed operating system in Java," in *Proc. 32nd ACM Technical Symp. Computer Science Education (SIGCSE)*, 2001, pp. 312–316.

[9] B. Atkin and E. S. Gün, "PortOS: An educational operating system for the post-PC environment," in *Proc. 33rd ACM Technical Symp. Computer Science Education (SIGCSE)*, 2002, pp. 116–120.

[10] W. A. Christopher, S. Proctor, and T. Anderson, Topsy -A Teachable Operating System 2002 [Online]. Available: http://www.tik.ee.ethz.ch~topsy/

[11] D. A. Holland, A. T. Lim, and M. I. Seltzer, "A new instructional operating system," in *Proc. 33rd ACM Technical Symp. Computer Science Education (SIGCSE)*, 2002, pp. 111–115.

[12] D. Hovemeyer, J. K. Hollingsworth, and B. Bhattacharjee, "Running on the bare metal with GeekOS," in *Proc. 35th ACM Technical Symp. Computer Science Education (SIGCSE)*, 2004, pp. 315–319.

[13] M. Goldweber, R. Davoli, and M. Morsiani, "The Kaya OS project and the μMPS hardware emulator," in *Proc. Int. Conf. Innovation Technology in Computer Science Education (ITiCSE)*, 2005, pp. 49–53.

[14] H. Liu, X. Chen, and Y. Gong, "BabyOS: A fresh start," in *Proc. 38th ACM Technical Symp. Computer Science Education*, 2007, pp. 566–570.

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

[16] Y.-P. Cheng, "Awk-linux: An educational operating system by program instrumentation," presented at the 6th IASTED Int. Conf. Computers Advanced Technology in Education, Rhodes, Greece, Jun. 2003.

[17] A. V. Aho, B. W. Kernighan, and P. J. Weinberger, *The AWK Programming Language*. Reading, MA: Addison-Wesley, 1988.

[18] G. Goebel, An Awk Primer 2006 [Online]. Available: http://www.vectorsite.net/tsawk.html

[19] A. Silberschatz, P. Galvin, and G. O. Gagne, *System Concepts*, 7th ed. Hoboken, NJ: Wiley, 2004.

[20] Y. Wiseman, "Advanced non-distributed operating systems course," *SIGCSE Bull.*, vol. 37, no. 2, pp. 65–69, Jun. 2005.

Yung-Pin Cheng received the M.S. degree in computer science from National Chiao-Tung University, HsinChu, Taiwan, R.O.C., in 1991 and the Ph.D. degree in computer science from Purdue University, West Lafayette, IN, in 2000.

He is currently an Associate Professor in the Department of Computer Science and Information Engineering at National Taiwan Normal University, Taipei, Taiwan, R.O.C. His research interests include software visualization, software design and analysis, software verification, and computer science education.

Janet Mei-Chuen Lin received the M.S. degree in computer science from Purdue University, West Lafayette, IN, in 1983 and the Ph.D. degree in computer science from Northwestern University, Evanston, IL, in 1989.

She is currently a professor in the Graduate Institute of Information and Computer Education at National Taiwan Normal University, Taipei, Taiwan, R.O.C. Her research interests include programming instruction at K–16, gender issues in computer science education, and knowledge management in education.