2006-889: USING LINUX KERNEL MODULES FOR OPERATING SYSTEMS CLASS PROJECTS

Timothy Bower, Kansas State University-Salina

Tim Bower is an Assistant Professor of Computer Systems Technology in the Engineering Technology Department of Kansas State University at Salina.

Using Linux Kernel Modules for Operating Systems Class Projects

Abstract

Instructors of operating systems classes have long desired to incorporate programming projects into the class that will give the students an appreciation for the source code of the kernel of a real operating system. Unfortunately, this lofty goal becomes difficult to effectively implement in practice. This paper reviews several approaches and environments for operating systems programming projects. A new approach involving Linux kernel modules and source code reading is described as a means to supplement other programming projects.

Introduction

In an operating systems class, we want students to gain an understanding of the internal data structures and algorithms used in real operating systems. As such, operating systems classes always include a heavy lecture component to expound on such topics as common operating systems architectures, device and I/O management, process management, memory management, synchronization, and file system management. However, lectures alone are not able to illustrate these principles in action. Students need some form of personal exploration to investigate how the concepts and algorithms are implemented. The logical solution is that students should study the source code of real operating systems and modify or instrument the source code to allow for closer understanding.

Unfortunately, operating systems class projects which use real operating system kernel code present many challenges. First of all, not all operating systems have source code available. Only open source operating systems such as Linux, MINIX and the BSD variants have source code available. Secondly, the source code for a real operating system is huge and formidable. It is challenging for even experienced programmers to wade through the volume of code to find the code related to the implementation of some algorithm, modify the algorithm such that the operating system still works and provides interesting insights about the operating system design and implementation. For a beginning programmer, it would be more frustrating than instructive. Thirdly, it can be administratively difficult to allow students to modify the kernel of an operating system. The process of compiling and installing a new kernel to support the wide variety of hardware and options that modern operating systems support is difficult and time consuming. It requires root level privileges; and worst of all, it risks damaging the installed system.

Modern Linux systems support a mechanism called loadable kernel modules that offer an interesting alternative. Kernel modules effectively allow for an addition to a running kernel in real time. The code for the kernel module must include a few basic components to facilitate the loading and removing of the module, but can be a fairly small and simple program. An obvious advantage is that students only need to work on and modify a small amount of code, which is isolated from the kernel's source code. When the module is loaded, it is linked to the executing kernel image, as if it were part of it from when the computer was first booted. Kernel modules cannot modify or replace existing functions in the kernel, but can add new functionality and can read and modify, if needed, any exported global variables and data structures. The ability to read the kernel's global data makes them ideal for student projects that examine global data

structures to more closely observe the behavior of the operating system. These projects typically only read the data, so the stability of the system is maintained.

With the introduction of the bachelor degree in Computer Systems Technology, Kansas State University at Salina offered an operating systems class for the first time in the fall 2004 semester. From the initial planning of the class, the laboratory programming projects were a primary concern. It was felt that programming projects using real operating systems would be perceived as more relevant and useful to students than projects which use a simulation environment. It was felt that the lab activities should involve some level of programming. However, it was desired to not have open ended programming assignments that would frustrate students, especially those with limited experience with Unix/Linux and programming in C. The primary programming experience of the expected student population was known to be with Java, Visual Basic and some C++ in a Windows environment. However, it was felt that exposing students to C and Unix/Linux would be a valuable addition to their educational experience. To cover the most material with lab activities, a preference was to have several small projects rather than a few larger projects. During the first semester the course was offered, two projects using Linux kernel modules were used. The idea for using Linux kernel modules started with a suggested lab assignment from Gary Nutt's book on kernel project's for Linux.[6] The lab introducing students to Linux kernel modules was enhanced with information from the Linux Kernel Module Programming Guide,[1] which is available on the Internet. Based on information available in the Linux Kernel Module Programming Guide, a device driver lab was also developed. For the second semester that the course was offered, in the fall 2005 semester, it was decided to take advantage of the kernel module's ability to read global data, so additional projects were developed which dealt with process control blocks and virtual memory management.

The Nature of Common OS Projects

Operating systems class projects usually fall into one of three categories:

1. Projects based on modifying the actual kernel of an operating system.
2. Projects where student write programs that run as user processes.
3. Projects using an operating system simulation environment.

Kernel Projects

As previously mentioned, projects based on modifying the source code of an operating system are ideal in terms of learning about the implementation of real operating systems, but are not the most practical. The primary downsides can be the programming difficulty of the projects and the administrative difficulties associated with compiling, installing and testing the compiled kernels on laboratory computers. As Nutt points out, when working with actual kernel source code, there is only time in a semester for a few projects.[6] The actual number of projects that can be completed in a semester would vary depending on the students' level of experience with programming in C and assembly language and working with Unix/Linux.

Perhaps the best resource for Linux based kernel projects is from Nutt's book.[6]

The MINIX operating system was designed specifically as a learning operating system. As such, it is significantly smaller and more manageable than other open source operating systems. MINIX is small enough that a compiled kernel can fit on a single floppy disk, which greatly

simplifies the administrative issues associated with booting the student compiled kernels. Tanenbaum's classic operating systems text book includes the full MINIX source code.[9] Unfortunately, MINIX source code, although smaller in size, is still quite difficult for beginning programmers and requires knowledge of both C and assembly language. MINIX also lacks the conveniences of a graphical user interface, which makes it more difficult for students with limited Unix experience to use.

In general, projects that modify source code of an actual operating system are not practical for a first operating systems class. In the preface of Nutt's operating systems text book, he reports that this was the unanimous opinion of operating systems instructors at a 1999 Operating Systems Design and Implementation meeting.[7] Linux kernel modules, however, offer a limited experience of working with real operating system source code that is practical for a first operating systems class.

User Space Projects

As Nutt points out, offering students the *external* view of real operating systems is the approach in the ACM/IEEE 2001 curriculum recommendation for first semester operating systems classes.[7] The approach here is to ask students to write user space programs in either Unix or Windows that will allow students to gain insight in the way the kernel works by using the facilities of the operating system, rather than modifying them. Several operating systems text books, such as Nutt's text[7], offer several suggested projects. Downey also offers suggestions on some interesting projects which focus on students using design concepts to execute the projects.[3]

These lab exercises clearly offer some value to the student and are much more manageable for the instructor than projects based on the *internal* view of operating systems. The obvious failing of this approach is that it does not show the students much about the implementation of real operating systems.

Simulation Environment Projects

Strictly speaking, this is a subset of user space projects because students write programs that run as user processes. A distinction is made because these projects focus on simulating a simplified operating system rather than exercising the operating system being used to run the program. Several environments have been developed to provide a framework for such projects. The most commonly used simulation environments seem to be Operating Systems Projects (OSP)[5], Nachos, and Ben-Ari Concurrent Interpreter (BACI)[2]. Appendix C of Stallings' text book offers a comparison of the various features of these environments.[8] Hu also describes an environment called Alchemy which offers some nice features.[4]

The basic approach is that the various parts of an operating system (CPU / process management, memory management, synchronization, and I/O management) are implemented as separate modules of a discrete time simulation. The instructor will typically remove part of the source code of one of the modules and provide the other modules as pre-compiled object code. The students must write the missing code for the module under study, compile the module and link it to the rest of the object code. When run, the correctness of the student's implementation is verified. Statistical data regarding the performance of the simulated operating system may also be calculated and reported.

The most compelling advantage of this approach is that students are allowed to implement the lowest level functionality of an operating system such as the process scheduler and virtual memory system. The approach also demonstrates how the implementation of various algorithms can impact the performance of the system. The drawback is that the system studied is not a real operating system. The students' code must account for the execution of the simulation rather than actual functioning of the operating system. Thus students are left to wonder how the code they wrote would compare to real operating system code.

As many instructors have observed first hand, these simulation environments are easier to work with than actual kernel code, but they still can be difficult for students to master.[3, 4, 8]

Linux Kernel Modules

As previously described, Linux kernel modules add to the functionality of a running Linux system. The common use of kernel modules in a Linux system is as device drivers. The kernel need not be compiled to support all the devices which might be present in a machine. The drivers for the hardware in the machine can be loaded when the system is first booted or as needed.

The most common way for users to invoke the kernel module code is with a read or write request of a special file. Special device driver files are an example of a file which can be used to invoke kernel module code. Another example of files that can invoke kernel module code when read are the files under the `/proc` directory. Linux maintains a file system (`/proc`) of files which when read will report various information from the kernel's global data. These are not actual files on the hard drive but are virtual files. When one of these files is read, (`cat /proc/foo`, for example) the `read()` system call invokes the appropriate kernel function to return the requested information in the same manner as it would invoke a function from the file system manager to read an actual file. The system has several such files by default, but a Linux kernel module can also create a new file under the `/proc` file system and provide the necessary read function. When invoked, the read function can report the value of global variables and data structures and any other desired information using a sequence of `sprintf()` function calls to write formatted string data to a buffer for eventual output.

The kernel module must include a small amount of "boiler plate" code to be executed when the module is installed and removed. This code is supplied to the students for most projects. For most projects, the main task of the student is to provide the read function which will report various global kernel data. The device driver project also asks students to study and modify code associated with `open()`, `close()`, `write()` and `ioctl()` system calls. To complete the projects, the students must gain an understanding of the data structures used in the kernel. In some cases, they also must make kernel function calls to obtain the data needed. For many projects, students also supplement the kernel module code by writing user space programs that cause the kernel's global data to reveal interesting insights about the functioning of the operating system.

Projects

In the second offering of the operating systems class at Kansas State University – Salina, the first two programming projects were user space projects designed to establish a minimal level of

competency with C and Unix and to establish basic concepts which would be used in later projects. Four projects involving Linux kernel modules were used.

The first project is to develop a simple shell using the `fork()` and `execv()` system calls to learn about process creation. This is Lab 2.1 as described in Nutt's text book.[7]

The second project is to write a program that reports the behavior of the system from the various files in the `/proc` file system. This project establishes an understanding of `/proc` files which are used in the kernel module projects. This is Lab 3.1 as described in Nutt's text book.[7]

The third project introduces Linux kernel modules. Using example programs available from the Linux Kernel Module Programming Guide[1], a sequence of simple modules are explored. The emphasis here is on understanding how kernel modules work and on working with the Linux system to compile, install, use and remove the modules. The first module is a simple "hello world" type program that students just install and remove while monitoring the system log file (`/var/log/messages`). The second module creates a read only `/proc` file. The source code for the first two modules is provided to the students. The third module asks the students to write a kernel module that reports the value of a global variable (`xtime`). The third module is the same as project four in Nutt's Linux kernel projects book.[6] However, after the students have studied the first two modules, the third module is a simple program to write. Finally, the students are asked to write a user space program to compare the system time returned from the `gettimeofday()` system call to the `xtime` global variable reported by the kernel module.

The fourth project deals with device drivers. The students expressed that they had a high interest in this lab because a device driver is something that they can visualize themselves developing at some time in their careers. Again, the students begin by looking at a sequence of modules obtained from the Linux Kernel Module Programming Guide.[1] The device studied here is a pseudo-device – a character device created in memory, rather than a physical hardware device. The first module creates a read only device. The next module creates a FIFO queue device that can be written to and read from. The second module also includes a function to be invoked from an `ioctl()` system call. The students are asked to write a small user program using `ioctl()` to exercise this part of device driver. The source code for the first two kernel modules is provided to the students. For the third module, the students are asked to modify the behavior of the second module such that if the device was written to consecutively, the subsequent writes would append to the queue rather than overwrite it. They are to implement this functionality using circular buffer. Largely because of a lack of experience with working with pointers in C, some students find this part of the project to be fairly difficult. For the fourth module, students are asked to modify the previous module to create a device driver that blocks the process using a wait queue when a process attempts to open a driver that is already in use, rather than exiting with an error code. The details of how to do this are shown in chapter nine of the Linux Kernel Module Programming Guilde.[1] So while this part of the project requires more difficult programming, an example is available to guide the students.

The fifth project relates to the process management discussion in the lecture part of the class. Students write their own kernel module to create a read only `/proc` file that reports data from the process control block data structures (`struct task`). The data reported for each process is similar to that of the Unix "`ps -f`" command with the addition of also reporting the state of each process. In addition to reporting data from the `task` data structure, students also use

linked lists to traverse the active processes. In the first pass, the module starts with the current process and follows the line of parent processes until the `init` process is reached. In the second pass through the active processes, the module looks at every process, but only reports data for processes which are not in the INTERRUPTIBLE state. To show at least one process, other than the current process, in an interesting state, students also write a simple program that creates a zombie process.

The sixth lab deals with virtual memory management. Students write their own kernel module to report on the virtual and physical memory used by the current process. The virtual memory used by the process is readily available from data structures pointed to by a variable in the current `task` data structure. Determining the physical memory used is a little more difficult because the page map tables must be read to translate virtual memory addresses to physical memory addresses. In some cases, the memory page containing the requested memory is not in physical memory but is swapped out to the disk. The kernel function calls that can be used to read the page map tables are pointed out to the students, but they must still write the code themselves to read the tables. Students also write a user space program that reads the `/proc` file, allocates a large amount of memory and reads the `/proc` file again and finally releases the allocated memory and once more reads the `/proc` file. From this exercise, they can see the dynamic nature of the page map tables and see how the system responds to page faults by moving memory pages out to the swap disk.

Other Projects Using Linux Kernel Modules

As was done in the last two labs discussed above, kernel modules can be used to examine any global kernel data relevant to whatever part of the operating system is under study.

Although not without compromising the stability and security of a system, it is possible to modify system data with a kernel module for educational purposes. Kernel functions can not be replaced with code from a kernel module, but system tables which include pointers to functions can be modified. One example of this is to replace the normal interrupt handler for a given hardware interrupt with a function defined in a kernel module. The Linux Kernel Module Programming Guide shows how to replace the keyboard interrupt handler.[1] This exercise was considered at K-State – Salina, but not done for a lack of time.

System calls may also be added or replaced using kernel modules, which greatly expands the possibilities for how kernel module code can be invoked form user processes. Replacing system calls could also provide a means for doing some interesting projects. However, as of the 2.6 version of the Linux kernel, the `sys_call_table` is no longer exported. This means that if system calls are to be used in projects, the kernel source code must be modified such that the `sys_call_table` is exported, compiled and the new kernel installed on the machines to be used. The administrative difficulties associated with replacing the kernel on the laboratory machines may prevent this option. Again, the Kernel Module Programming Guide shows an example of how to replace a system call with code contained in a kernel module.[1]

Administrative and Security Issues

Compiling Linux kernel modules is not an issue. The `make` command is used to compile the kernel module and link it to the needed object files from the kernel source code. See the Linux

Kernel Module Programming Guilde[1] for an example of how to structure the `Makefile`. It is necessary that the kernel source code be installed on the Linux system.

Not surprisingly, there are a few administrative issues and security concerns that must be addressed in order to allow a class of students to use kernel modules with laboratory machines. Root level privileges are needed to install and remove kernel modules. The device driver project also requires root level privileges to create the special device driver files. Perhaps the largest security risk comes from the need to make the device driver files ones that can be written to. By the default, the `mknod` command creates device driver files that are read only.

The `sudo` command was used at Kansas State University – Salina to give students limited root level privileges to use the commands that are needed to complete the laboratory activities. With the appropriate entry in the `sudo` configuration file (`/etc/sudoers`), users can execute a specific command with root level privileges. They are asked to enter their user password to complete the requested command, but do not need the root user's password. So, for example, to install a compiled kernel module object file named `chardev.ko`, a command of "`sudo /sbin/insmod ./chardev.ko`" would be used. (Note that the `.ko` extension is used to indicate that the file is a kernel object file that has been linked to the needed object files from the kernel source code. The file is still considered an object file, not an executable file. It only becomes part of an executable image when it is loaded and linked in real time to the running kernel.)

At Kansas State University – Salina, with the help of a very cooperative systems administration staff, the laboratory machines were configured to have the students all use the same login name (tc182, which is the lab room number) and password for the purposes of the laboratory activities. The most difficult part of the configuration was allowing the students to change the permissions of the device driver special files. For this, the students were required to place their device driver file in a special directory. The machines were further configured to delete all the files in this directory when each session ended. The additions made to the `/etc/sudoers` file is shown below.

```
tc182   ALL=/sbin/insmod
tc182   ALL=/sbin/rmmod
tc182   ALL=/bin/mknod
tc182   ALL=/bin/chmod go[+-][rwx] /home/tc182/labs/*
tc182   ALL=/bin/chmod [go][+-][rwx] /home/tc182/labs/*
tc182   ALL=/bin/chmod go[+-][rwx][rwx] /home/tc182/labs/*
tc182   ALL=/bin/chmod [go][+-][rwx][rwx] /home/tc182/labs/*
tc182   ALL=/bin/chmod go[+-][rwx][rwx][rwx] /home/tc182/labs/*
tc182   ALL=/bin/chmod [go][+-][rwx][rwx][rwx] /home/tc182/labs/*
tc182   ALL=/bin/chown \:users /home/tc182/labs/*
```

Teaching Experiences

To successfully complete each project, the students first need to understand the operating system concept studied. They must read and understand any source code provided by the instructor. They must also read and understand the Linux kernel source code to understand the data structures used and kernel function calls available. The latter step usually involves a significant

amount of research into the documentation available on the Internet about the Linux kernel, as well as source code reading. Each project is accompanied by a number of specific questions about the source code of the kernel modules and the Linux kernel. Students are required to write a detailed lab report for each project showing their own source code, program output and demonstrating an understanding of the topic studied.

The student response to using Linux kernel module projects was fairly positive. At the end of the fall 2005 semester, a survey was conducted to measure student response to the approach. Every student responded that they either agreed or strongly agreed with the following statements.

1. Using Linux kernel modules in the lab activities provided a good opportunity to learn about operating systems.
2. I liked the fact that by using Linux kernel modules, we developed code that integrated with the kernel of a real operating system.
3. Enough interesting projects can be done with Linux kernel modules to satisfy the laboratory needs for an undergraduate operating systems class.

Surprising information learned from the survey was that they uniformly did not like using Linux and that they thought the projects were difficult. These two responses were probably related to each other. Most of the class had little or no experience with Unix/Linux and found interacting with the shell's command line interface to be difficult. They also had limited prior experience using C. They needed to do independent research and source code reading to be able to write the needed source code and answer the required questions. So from an instructional perspective, the negative responses on the student survey about the projects being difficult were not necessarily disappointing.

The pedagogical advantages to using Linux kernel modules for operating systems class projects include:

- Projects can be closely related to material covered in lectures.
- Several small projects can be completed, rather than only a few large projects.
- Projects are structured and doable for a first operating systems class. That is not to say that students are not challenged, but the volume of code they work on is a manageable size and it is clear to them that they are not being asked to work on an open ended programming project that is beyond their capability.
- Projects make use of a real operating system which increases the students' perception of the projects being relevant. It also boosts the students' confidence by demonstrating that they are capable of working on code which integrates with the rest of the operating system.
- Projects require students to do independent research and source code reading to understand the implementation used in a real operating system.

Conclusion

The overall experience of using Linux kernel modules for operating systems class projects was quite positive. Since the class at Kansas State University – Salina has only been offered twice and kernel modules were used both semesters, it not possible to fully assess the impact on student learning. As the instructor, it seemed clear that the kernel module projects helped the students to understand the material covered in lectures. The student’s perception of the value of

kernel module projects was also high. A plan for a future semester is to combine the use of Linux kernel module projects with simulation environment projects.

Bibliography

[1] Burian, Michael & Salzman, Peter Jay & Pomerantz, Ori. 2005, The Linux Kernel Module Programming Guide. The Linux Documentation Project web site: http://www.tldp.org/LDP/lkmpg/

[2] Bynum, B., & Camp, T. 1996, After You, Alfonse: A Mutual Exclusion Toolkit. Proceeding of the 27th SIGCSE Technical Symposium of Computer Science Education. 170-174.

[3] Downey, Allen. 1999, Teaching Experimental Design in and Operating Systems Class. Proceedings of SIGCSE 1999, 316-320.

[4] Hu, G. 1994, A Simulated Hardware for an Operating System Course Project. Computer Science Education. 5(2), 45-62.

[5] Kifer, Michael & Smolka, Scott A. 1991, OSP: An Environment for Operating System Projects. Reading: Addison Wesley.

[6] Nutt, Gary. 2001, Kernel Projects For Linux. Boston: Addison Wesley Longman.

[7] Nutt, Gary. 2004, Operating Systems, Third Edition. Boston: Pearson Addison Wesley.

[8] Stallings, William. 2005, Operating Systems: Internals and Design Principles, Fifth Edition. Upper Saddle River: Pearson Prentice Hall.

[9] Tanenbaum, Andrew S. & Woodhull, Albert S. 1997, Operating Systems: Design and Implementation, Second Edition. Upper Saddle River: Prentice Hall.