TEACHING LINUX BASED OPERATING SYSTEM

Warren Ebell
Computer Science Honours
University of Cape Town
webell@cs.uct.ac.za

Liesl Lohlun
Computer Science Honours
University of Cape Town
llohlun@cs.uct.ac.za

Yat Pan Ng
Computer Science Honours
University of Cape Town
ypng@cs.ucta.c.za

ABSTRACT

The teaching of operating system internals at UCT does not take a practical view of the operating system. Students are not given the opportunity to gain any experience with source code, and the inner functions. To decrease the impact of testing, a minimized version of the operating system would be ideal.

The Linux operating system presents opportunities to alter source code, but the amount of information available to prospective users is limited. Also there is a means to implement a minimalist version of the kernel.

Sections of interest are the scheduler, the file system, and the virtual memory manager. The goal was to provide detailed information about each section, as well as some means of altering it.

Categories and Subject Descriptors

D.4 Software - Operating Systems

Technical Report Number

CS04-11-00

General Terms

Operating Systems, Scheduler, Virtual Memory Manager, File Systems.

Keywords

Linux, process scheduler, O(1) algorithm scheduler, loadable module, kernel, virtual memory sub-system, paging algorithm, Virtual File System.

1. INTRODUCTION

The field of computer science has always been a two sided coin: there has always been a need for the theoretical side of computational science to push the boundaries of the science, and there has always been the practical side which has always tempered the theoretical side by limiting the implementation of the theory, and also providing a “what works in the real world” view to the science. These two sides are always weighed up in the teaching of computer science; some institutions believe that a theoretical grounding is better, while others take a more practical stance. At the UCT Department of computer science the curriculum is well balanced over the majority of the fields of study in computer science, with the exception being operating systems. This is partially due to the fact that facilities to work with actual operating systems are limited and no-one wants to have a communal lab being brought down by students altering OS source code and not succeeding.

This paper will look at an alternative way to teaching a practical based course in operating systems using Linux as a base. This can be accomplished by making use of current resources, and does not require a host of new labs to be set up just for this course. The operating system will be compiled onto a bootable floppy disk, and this will prevent the bringing down of the lab environment.

Some of the major components of the operating system will be presented as topics of investigation, and these will give learners the opportunity to work with the source code of an operating system, and gain practical knowledge to accompany the theoretical knowledge gained in previous courses.

As mentioned the operating systems course in third year takes a high level approach to the operations of the operating system. Students get very little exposure to the inner workings of the OS, they are just told what happens and have to accept that the theory being presented is actually implemented in the manner presented.

This extends to many of the kernel components, and students have very little exposure to these “real world hacks” that give the OS the necessary performance, but still keep it relatively theoretically correct. The motivation is to give students exposure to these, so as to extend their theoretical knowledge with practical application.

2. BACKGROUND AND MOTIVATION

Operating systems are an important part of the computer science field, and the teaching of them is not done on a very practical level. A survey of operating system professors was conducted by Addison-Wesley and the general consensus was that they want to get students to be involved in programming projects, but there is no real tool to help them with this.

A lot of work has been done to come up with a teaching tool that would allow students to have a strong feel of the core mechanisms of an operating system.

In the preface of his book Gary Nutt provides a realistic view of the teaching of operating systems:

“There are only a few widely used commercial operating systems. While studying these systems is *valuable*, there are practical barriers to experimenting with any of them in the classroom. First, commercial operating systems are very complex since they must offer full support to commercial applications. It is impractical to experiment with such complex software because it is sometimes difficult to see how specific issues are addressed within the software. Small changes to the code may have unpredictable effects on the behaviour of the overall OS. Second, the OS software sometimes has distinct proprietary value to the company that implemented it".

Another weak point of teaching operating systems on a practical level is the lack of documentation. For example there is very little documentation regarding the inner workings of the Linux kernel. Someone who is new to the kernel is "politely" told to RTFS[1]. This presents a problem, as the source is at best confusing to read, and takes a few attempts before it is understood.

By providing students with easy to understand documentation and a simple way to get exposure to the inner workings of an operating system would make for a course that would give the learners the necessary exposure to the practical implementation of an Operating System.

The motivation for this practical exposure to operating systems is that students only get theoretical exposure to operating systems in the third year course, with limited practical tutorials. The most complex of these tutorials is writing a simple device driver.

3. APPROACH AND METHODS

The first hurdle that a practical course in operating systems faces is the source code that is going to be used. As some of the operating systems out there are propriety there are a limited number of options available to use. The solution to this problem is to make use of one of the open source operating systems, as the source code is freely available for download, and there are people available to answer questions for "newbies[2]" via newsgroups or forums. We chose to go with the Mandrake Linux Community 10 distribution as it is free, and it includes all the necessary utilities for us to do our work (although any of the Linux distributions would have worked).

Our project supervisor initially recommended the 2.4 series kernels to work on as there is much documentation about it, but changed this to the 2.6 series as the 2.4 series is quite old, and there are many new functions in the 2.6 series that could prove to be useful. The core functionality remains the same between the two versions, so changing was not much of a hiccup to the progress of the project.

Another problem is the facilities available. There would be very little chance to have a dedicated lab available for just one course. Using a communal lab could present the problem of having some of the machines rendered useless by an attempt to run a kernel that does have some flaws.

To overcome this problem we suggested the use of dual-booting the machines. Having more than one operating system on each machine would allow students the opportunity to have one stable environment to use for day to day tasks and practicals, while the other environment could be used for testing kernels.

"Testing? What's that? If it compiles, it is good, if it boots up it is perfect" – Linus Torvalds

This method initially seemed plausible, but after much discussion was not implemented as a better method was found. This new method involved compiling the kernel onto two floppy disks using the Pocket Linux guide [1]. This method provides a means for the students to compile their own kernels onto a medium that would not cause lab interruptions, but would still give them an opportunity to test their work. If the kernel then compiled and booted correctly, it could be implemented on the Linux machines as a different boot option in the boot menu (using LILO or GRUB).

In terms of sections that could be learnt, three of the major components that are common to most major operating systems were chosen for further investigation. These sections were the Linux Virtual Memory sub-system, the process scheduler, and the file system. With these three sections a learner would gain exposure to some of the important core components of an operating system.

Initially it was envisioned that each of these sections could be abstracted out of the kernel, and give the learners an interface to the kernel so as to minimize the amount of kernel code that would need to be understood to write a working module. The Linux operating provides for the loading of kernel modules that extend the functionality of some of the core kernel functions, one example being file system modules.

Research on scheduling algorithms has been a popular topic ever since multi-tasking was first introduced. Many new algorithms have been developed since, but there has been little change in the schedulers used by commercial operating systems. One of the main obstacles that scheduler developers face is the implementation of a scheduling policy in a standard OS: developing and implementing a scheduling policy require two different kinds of expertise. Therefore a scheduler developer needs both to master kernel hacking and to be knowledgeable in the scheduling research field.

There were not many papers or articles that specifically deal with how to write a loadable that can replace the default scheduler in Linux 2.6 or for any version of Linux for that matter, but there is a guide [7] that provides high-level instructions on how this can be done in Linux 2.2.14.

Another related piece of research done was DWCS [8]. DWCS stands for Dynamic Window-Constrained Scheduling. Originally, DWCS was designed to be a network packet scheduler that limited the number of late or lost packets over network traffic stream, but later it evolved into a process scheduler for Linux. DWCS can be configured to run as an earliest deadline-first (EDF), static priority or proportional share scheduler. DWCS also has the desirable property of attempting to guarantee no more than x deadlines are ever missed in each window of y consecutive deadlines.

Bossa is a kernel-level event-based framework to facilitate the implementation and integration of new scheduling policies. It uses a domain-specific language (DSL) that provides high-level scheduling abstractions that simplify the implementation and

[1] Read The Freaking Source.

[2] Newbie – someone who has little or no knowledge of a subject

evolution of new scheduling policies. A dedicated compiler checks Bossa DSL code for compatibility with the target OS and translates the code into C. However, Bossa currently only supports 2.4.2 and because of the major different in the scheduler source-code between 2.4 and 2.6, it was deemed to be un-portable to the 2.6 kernel hence unusable in the project.

4. COMPONENTS

4.1 Linux Virtual Memory sub-system

The main area of focus in the VM was with the paging sub-system. This was chosen because it has a direct influence in the performance of the operating system, and poor implementations could be very easy to pick up [2, 3]. Also the implementation of the paging algorithm does not follow the theory of the Least Recently Used strictly [4], and would therefore give the learners an opportunity to see how "real world" performance influenced the implantation of the theory.

The Least Recently Used algorithm is implemented in a way so as to minimize the overhead [4], while still giving the best possible performance. The page Table contains a reference bit that is set when the page is accessed. If a page fault occurs, all of the reference bits are checked. If the bit is zero, that page is marked for reclamation. Once all the pages have been checked the reference bits of those not marked for reclamation are set to zero and the pages marked for reclamation are swapped out. This method works well in the real world, and is a very close approximation of the Least Recently Used algorithm, but does have its flaws.

In the extreme case that two page faults occur right after one another, all of the pages will be swapped out. The process happens like this: after the page fault all of the reference bits are set to zero. The second page fault then occurs, and as none of the pages have been referenced since the last page fault, all of them will be marked for reclamation, and will be swapped out. This can lead to a form of thrashing, but only in an extreme case.

A solution to this problem would be to implement a frequency count to work in conjunction with the LRU algorithm. This would keep the pages that have a greater frequency of use from being swapped out, even if they have not been referenced since the last page fault. The frequency count would be reset when the page is swapped out. This implementation of the Least Frequently Used algorithm on top of the LRU would solve the double page fault problem, but any algorithm could be implemented instead of the LFU.

Another option would be to increase the number of bits used to check the number of references a page receives [5, 6]. This would be a better implementation as a better indication of the history of page usage is kept, while still keeping performance at an acceptable level.

4.2 Process Scheduler

A scheduler affects the overall feel of a system. Whether this is the interactivity of a desktop client or the throughput of an application server, the operating system installed usually only employs one algorithm. This algorithm will then need to perform well in both cases.

In a traditional O(n) scheduler, calculating time slice often requires us to loop over each task. This calculation is usually done when the time slices of all the currently running processes have all been expired. Recalculation of each task's time slice will then require a loop of order n, and the priority of each task and other attributes are subsequently used to determine the time slice given to the task. Not only does this approach face the danger of scaling to a O(n) algorithm for n task, but locking must also be done to ensure the task list is not tempered with during recalculation

The new scheduler uses a fixed range of priorities in a doubly linked list to ensure O(1) scheduling time. This structure is the result of the hybrid between the famous round-robin and first-in-first-out. The data structure used to store the list of runnable processes is the runqueue. Each runqueues contains two priority arrays, the active and the expired array. The struct prio_array is responsible for implementing this array. It is defined in /kernel/Sched.c [9] and it is crucial in order to provide the O(1) scheduling. Each priority array consists of one queue of runnable process per priority level, creating a structure similar to that of a double link list with one list having fixed length.

The priority array also contains an array of longs called bitmap, and the bits in this long array are used to keep tract of which queue is empty and which is not. It is the fixed length (140) of the runqueue and the bitmap that guarantee the new scheduler its O(1) efficiency.

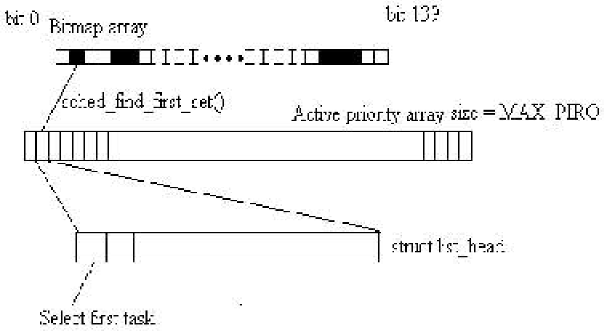

Figure 1. An illustration of the process taken to select the next task to run. [12]

The interactivity of a task is not something that is known intuitively by the scheduler. It uses a heuristic that correctly quantify the interactivity of a task by checking whether a task is I/O bound or processor bound**.** This metric has the advantage of not being vulnerable to abuse. If a heavily I/O bound task spent a long time sleeping but also quickly used up its entire time slice, it will not be given a large bonus. The idea is not just to award interactive task but also punish processor bound task. A newly started process quickly receives a large sleep_avg to allow for immediate response. Later the penalty or bonus will also affect the dynamic priority based how much a process hogs the processor.

One of the advantages of the new scheduler is the improved scalability that it provides. Each processor has its own separate runqueue and locking, and hence each cpu only maintains a list of its own processes and the scheduler schedules each runqueue separately. In a SMP system, this will probably create an imbalanced load amongst different processors. Some runqueues might be longer than other and some might even be idle while others suffer from process starvation. Intuitively, this problem requires a global scheduling mechanism, but it is solved with the load_balance() function which is also an "one per processor function".

This function compares the runqueues from different cpus and find imbalance amongst them. If there is such an imbalance, it will attempt to pull suitable tasks from the busy processor to an idle or a less busy one. The idea is to make sure all runqueues have more or less the same amount of processes. It is called by schedule() when the runqueue is empty, and it is called by the timer every 1 ms when the processor is idle or else every 200 ms.

The design of the module [11] that allows users to switch between default and custom scheduling policy were made possible by adding additional system calls [10] and using them as stubs between the kernel and kernel modules. While this design might not be security wise, it is does get the job done. Keep in mind that the modularization is done for educational purpose, so the tools or the module is not meant for commercial use.

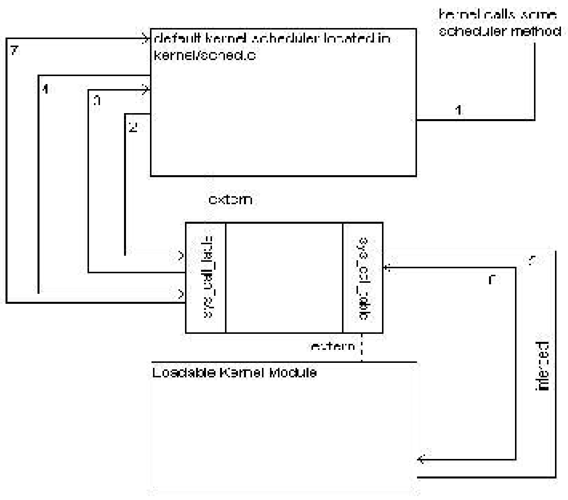
kernel calls some scheduler method
default kernel scheduler located in kernel/sched.c
7
4
3
2
1
extern
sys_call_table
sys_call_table
extern
Loadable Kernel Module
6
5
infected

Figure 2

1. Some code in the kernel calls anyone of those methods

2. The method calls the system call getSchedConfig() to check whether the original or the hacked version should be called.

3. getSchedConfig() returns an answer.

4. If a hacked version is needed then {*method*}_mod() system call is called.

5. The module in SchedMod.c will intercept this call, as it has already replaced the {*method*}_mod() address on the sys_call_table with an address of its own.

6. After calling the hack version, the return value (if there is one) is return to the {*method*}_mod() system call.

7. {*method*}_mod() just returns the return value to method in sched.c.

4.3 File Systems

When the Linux kernel needs to access a file system, it uses a file-system-type independent interface, which allows the system to carry out operations on a file system without knowing its construction or type. One of the most important features of Linux is its support for many different file systems. Since the kernel is independent of the file system type, it is flexible enough to accommodate future file systems as and when they become available.

In Linux, the separate file systems that the system may use are not accessed by the device identifiers (such as drive number or drive name), but instead they are combined into a single hierarchical tree structure that represents the file system as an entire, single entity. Each new file system is added into this single file system tree as it is mounted. All file systems, no matter what type are mounted onto a directory and the files of the mounted file system cover up the existing contents of that directory. This directory is known as the mount directory or the mount point. When the system is unmounted, the mount directory's own files are once again revealed [13].

Linux allows you to use loadable modules for all the file system types. These software modules can either be linked to the kernel being booted or compiled in the form of loadable modules. In the case where file systems are built as modules, they can be demand loaded as they are needed or loaded by hand using, "insmod". Whenever a file system module is loaded it registers itself with the kernel and unregisters itself when it is unloaded. Each file system's initialisation routine registers itself with the Virtual File System and is represented by a `file_system_type` data structure which contains the name of the file system and a pointer to its VFS superblock read routine [14].

5. FINDINGS

Using the Pocket-Linux guide to compile a kernel was more than just a step-by-step process. The main problem was getting hold of all the correct source code, and then ironing out all the compile time errors. The guide recommends stripping all the non-essential parts of the kernel so as to keep the size down, but in our case some of these non-essential components could be used to better understand the kernel functionality. There are also some components that are completely removed in the guide that would aid in learning, such as access to hard disks, to access benchmark files, or creating logs that could be viewed at a later stage in a stable environment.

The modularization of the VM sub-system proved to be too much of a challenge, as it is too extensively linked into other core components of the kernel. Some progress was made in deciphering the little documentation that is available, and comments were added to the source code in order to easy the understanding of the functionality, and the purpose of the methods.

The 2.4 series of kernels were easier to understand, especially the earlier ones, as the paging algorithm was still implemented in software, and this aided the understanding of the later kernels in the 2.4 series, as well as the 2.6 series. The differences between the two major versions is that the 2.6 kernels have more than one swap-out daemon (one daemon per memory node), which is not all that much of a change. All that is required is that the previously global variables be moved so that there would be one copy of each in each zone of the nodes

In order to compile a module the path of the Linux kernel source code is required, and since the compressed source is already 35MB and 120MB when decompressed, it is pretty unlikely that we can provide the a modifiable teaching tool based on just floppy disk. Alternatively we can, mount a USB flash drive, and direct the path of the Linux source code there.

Majority of all the kernel source code were written in C, the rest in assembler, and this might pose a great challenge for students, as the C language is not part of their syllabus. Even for someone who is proficient in C, there are many macros and extern methods that are defined outside the current .c file that one might be looking at.

The implementation of the scheduler module requires a pretty in-depth knowledge in C, and it also requires a very skillful and advance programmer. When someone is writing a normal main() program, a syntax error means a quick shout from your gcc compiler and a logic error means a core dump or segmentation fault. But since kernel module runs in kernel space, your little syntax error will probably only be discovered halfway through your kernel recompilation which takes about 10 minutes. Any logical error in the module will either freeze the entire system or reboots the system without a warning, neither is desirable.

The following summarizes the major findings when attempting to modularize the process scheduler.

- The new O(1) Scheduler is a hybrid of the two tradition scheduling policy RR(round robin) and FIFO (first in first out).
- Understanding the current O(1) scheduler is already an appealing educational exercise.
- Writing the methods that will replace the default ones will require the student to be skillful in C and to have good knowledge of Linux internals.
- While writing a module and system-calls to abstract out the scheduler is a good idea, it is not a strategy that I recommend to encapsulate ALL parts of the operating system.
- Pocket Linux alone will not be able to produce a developing environment for kernel editing, modifying and testing.
- Using default system calls can also change the current behavior of the scheduler to a certain degree.
- If the 2.4 version were used, either Bossa or DWCS can be used to modularize the scheduler.

Writing a file system requires following a few basic steps, namely: Register the file system, write a function for reading the superblock (the data structure that holds information about each mounted file system) and for setting up the superblock structure and most importantly the superblock field, write various super_operations functions, write various inode_operations functions and finally write various file_operations functions [15].

Due to the fact that the current implementation of Linux allows you to use loadable modules, adding a new file system to the kernel is made relatively easy. The kernel does not need to be recompiled every time a change is made. The new file system can be written and compiled separately and then loaded into the kernel if and when it is needed.

6. CONCLUSION

To give students a more practical course in operating systems requires a stable OS that also allows some form of configurability. Previous work found that the Windows CE platform was not suitable for this application. The Linux kernel does conform to these requirements, and can also be implemented in a minimalist manner which is also desirable in terms of ease of testing.

The Linux kernel is available for free, unlike some proprietary systems, and there is a community of knowledge available in the form of online guides, numerous books as well as message boards. Although these seem like a plentiful source of knowledge, there are some sections of the kernel that are only fully understood by a small number of individuals that are involved in the development of kernels, and this could be seen as a detracting factor.

As we have shown, the Pocket Linux Guide is easy to follow and the end product is a form of throw-away testing which would not influence the uptime of laboratory workstations due to buggy code. The “pocket-linux” also provides a fast method by which kernel compilations can be tested. There would still be a need for some form of Linux specific laboratory for the students to do their development, but the need for testing specific machines is eliminated.

One negative that became apparent during the project was the lack of knowledge of the C programming language. Although there are many similarities between C and C++, some of the conventions needed to be investigated to be fully understood. If the operating systems course were to go ahead, an introductory into the C programming language would be needed before a student could progress onto working with the source code.

The Linux kernel is after all monolithic and hence heavy coupling, even after the scheduler was abstracted as a module and modifications were made, it would be extreme difficult to tell what went wrong should an error occurs. By just knowing one part of the OS, it is often difficult to see what other parts are connected or indirectly connected to it, therefore it would be wiser to look at a simpler and earlier version of Linux in the future but with specify documentation or books already obtained.

With the increasing popularity of the Linux operating system, choosing it to be used as a base from which to teach a practical operating systems course would not only benefit the students in the future, but also the open source community, as there could conceivably be more individuals with experience in the inner workings of the operating system.

7. ACKNOWLEDGMENTS

Our thanks to:

Matthew West, System Administrator in UCT Computer Science Department, for helping us in providing background information about Linux kernel.

Prof. Ken MacGregor, our supervisor, for guiding us along the way and providing background materials.

8. REFERENCES

[1]. David Horton. Pocket Linux Guide. http://my.core.com/~dhorton/linux/pocket/ 2004

[2]. Mel Gorman. Understanding the Linux Virtual Memory Manager. Prentice-Hall,2004.

[3]. Abhishek Nayani. Memory management in Linux – Desktop companion to the Linux source code, 1994.

[4]. Gary Nutt. Operating systems second Edition. Addison Wesley, 2002.

[5]. Rodney R. Oldehoeft, Maekawa Mamoru and Arthur E. Oldehoeft. Operating Systems, Advanced Concepts. Benjamin/Cummings Publishing, 1987.

[6]. Andrew S. Tanenbaum. Modern Operating Systems, 2nd Edition. Upper Saddle River, NJ: Prentice-Hall, 2001.

[7]. Scott Rhine, Hewlitt-Packard Company, http://linux.ittoolbox.com/documents/document.asp?i=1117, 3 Mar 2000

[8]. Rich West, http://www.cs.bu.edu/fac/richwest/dwcs.html, Jun 2004

[9]. Panagiotis Christias, http://unixhelp.ed.ac.uk/CGI/man-cgi?sched_setscheduler+2, Jun 2002

[10]. Worcester olytechnic Institute, http://fossil.wpi.edu/docs/howto_add_systemcall.html, Jun 2004

[11]. Jay Salzman, http://www.tldp.org/LDP/lkmpg/2.6/html/, May 2004

[12]. Robert Love, http://www.samspublishing.com/articles/article.asp?p=101760, Nov 2003

[13]. Freeos.com, http://www.freeos.com/articles/3838/, March 2001

[14]. A. Rubini, The“Virtual File System” in Linux, May 1997, http://www.linux.it/kerneldocs/vfs/vfs.html

[15]. Writing a Linux FileSystem Module, July 2001, http://www.cise.ufl.edu/~ppadala/publications/fs/slide001.html