The Kaya OS Project and the μMPS Hardware Emulator

Michael Goldweber
Xavier University
mikeyg@cs.xu.edu

Renzo Davoli
Universitá di Bologna
renzo@cs.unibo.it

Mauro Morsiani
CINECA Interuniversity Consortium
mmorsian@cineca.it

ABSTRACT

Ideally, the most meaningful learning experience for students in an undergraduate OS course would be to develop fully-functional OS's on their own. This can be accomplished using μMPS, a hardware emulator for a pedagogically undergraduate-appropriate hardware architecture, along with Kaya, a specification for a multi-layer OS supporting multiprocessing, VM, thread synchronization, external devices (disks, terminals, tape, printers, and network interfaces) and a file system.

Traditional OS projects like Nachos[3] or OS/161[9] provide students with a significant starting code base. Students then modify existing OS modules or add new ones. With μMPS/Kaya students undergo an innovative and pedagogically different experience of starting only with a hardware emulator (i.e. no initial OS code base for students to build on/replace) and ending with a completely student written OS capable of running student written C programs.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: [Computer Science Education]; D.4.7 [**Operating Systems**]: Organization and Design—*Hierarchical Design*

General Terms

Design

Keywords

Operating Systems, Education, Hardware, Emulation

1. INTRODUCTION

Operating systems are worthwhile objects of study because of the concurrency they both implement and exhibit, and the complexity they both hide and possess. In a very real sense, the study of operating systems is a microcosm of computer science in general; algorithmic problem solving and the taming of complexity through abstraction. Therefore, the OS course is a required course in most undergraduate programs, often providing a capstone-like experience[11]. Unfortunately, instructors of the OS course are limited in their choices for comprehensive student projects.

We assert that the goal for the undergraduate OS project is not to develop future OS authors, but talented overall computer scientists. Hence the minutiae of individual component/algorithm performance is secondary to a comprehensive view of the whole and seeing operating systems as an application of both concurrency and abstraction to manage complexity. Therefore, given this perspective, the most meaningful learning experience in an OS course would be for students to develop a complete OS on their own. In addition to focusing on the alternative algorithms for a given component (e.g. CPU scheduling, deadlock detection, or paging), students who write their own OS gain an appreciation for the overall design and architecture of operating systems and how each component both subtly and not so subtly interacts with each other. This task not only illustrates how to implement concurrency, but how to both harness concurrency and manage complexity. Of necessity, this student written OS would not be overly sophisticated, but would hopefully be complete. By complete, we mean that all the primary components of undergraduate study, as defined by the table of contents in any popular undergraduate OS textbook[19, 21, 22], are present; scheduling/multiprocessing, deadlock handling, process synchronization, device management, virtual memory, files, etc. By unsophisticated we mean that much of the code needed for robustness and functionality for a wide variety of environments/users necessary for commercial operating systems is missing.

In order to facilitate the creation of a student written OS, one needs a hardware emulator[5]. Hardware emulators are used to eliminate the burden of working on a bare machine, which, given the time frame of a single term, is outside the scope of an undergraduate's ability. Hardware emulators can provide sophisticated development, user interface, testing, and debugging environments unavailable when working on a bare machine. Furthermore, the architecture of real commercial CPUs in order to achieve super high speed operation can be overly complex in their detail, obscuring the basic underlying features and unnecessarily complicating students' understanding. μMPS is a new hardware emulator designed specifically for supporting the creation of undergraduate student written operating systems, which while based on the MIPS R3000 exhibits a pedagogically advantageous (i.e. simple and uncluttered) CPU architecture.

Unfortunately, a hardware emulator such as μMPS is, regardless of its design goals, by itself insufficient to insure its widespread use. Even for those with many years of experience teaching OS, the creation of the necessary accompanying curricular materials, in essence the design of a complete though unsophisticated OS broken down into a series of realistically appropriate student assignments, is a daunting task. The Kaya OS project is a set of curricular materials for the creation of a completely student-written unsophisticated, though complete, OS using the μMPS hardware emulator.

2. THE μMPS EMULATOR

The pedagogic philosophy embodied by μMPS/Kaya is not completely new. The Cornell Hypothetical Instruction Processor[1] (CHIP) and the Hoca OS specification[2] designed for CHIP was the first OS courseware system to take this approach. Unfortunately, due to a lack of sufficient dissemination CHIP/Hoca never saw widespread use. Currently it is too out of date for continued use.

The MPS[18] emulator was created expressedly to be a replacement for CHIP. While CHIP was based on the architectures of the PDP-11 and IBM S/370, MPS faithfully emulates the architecture of the MIPS R3000. Like CHIP, MPS also possesses a sophisticated user interaction and debugging facility. MPS also emulates a rich set of memory-mapped character-based (i.e. terminals and printers) and DMA supporting block-based (disks and tapes) external devices and a real network interface (available through VDE[4]). Furthermore, not only was the Hoca OS specification –which itself was based on the specification of the THE OS[6]–updated to work with MPS (Tina[17, 16]), but a message passing-based OS specification, AMIKE[15] was produced as well.

Not surprisingly, in hindsight, MPS/Tina, due to the complexity of MIPS' virtual memory management was, as determined via class testing, unsuitable for undergraduates. In the MIPS architecture, virtual memory is always on, all address translation is performed through a small fixed size TLB, and hence even the OS maintained page tables for itself and user processes are kept in virtual memory. Furthermore, the physical address space for the kernel and its data structures are permanently disjoint from its virtual address space. While these RISC-design features allow for an extremely fast CPU they complicate introductory students' understanding; in particular with the circularity of an OS always running with VM on and whose page tables are kept in virtual memory.

A design goal of μMPS was to implement a virtual memory management subsystem that more closely matched the conceptual description found in introductory OS texts[19, 21, 22]. More specifically:

- A VM bit was introduced into the STATUS control register allowing for address translation to be turned on and off.
- Formal segment table and page table formats were introduced.
- If the TLB does not contain the appropriate entry, μMPS, via the appropriate segment and page tables locates the missing entry and inserts it into the TLB.
- All of the segment and page tables, both for the kernel and for user processes are stored in permanent physical locations. This eliminates the circularity of having the OS data structures for supporting virtual memory being kept in (and hence managed by the) virtual memory.
- The cross compiler that accompanies μMPS compiles the student OS to reside in a different segment than the one the user programs are compiled to reside in.

Since VM can be turned off students can appropriately focus on introductory tasks (e.g. scheduling, interrupt handling, device management) prior to dealing with memory management. Furthermore, short OS "critical regions" are more easily handled; simply disable both interrupts and VM for the "critical region."

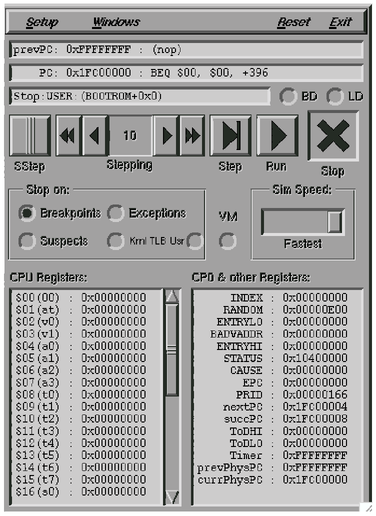
When VM is on, address translation still uses the TLB, but when a TLB-MISS event occurs, the system will locate the correct page table by examining the segment table and then search the indicated page table. If successful, the TLB is updated, otherwise a PAGETBL-MISS exception is raised. It is the responsibility of the kernel author to maintain the segment and page tables for the kernel and for all user processes.

Finally, the (virtual) segment the kernel is compiled to reside in is such that one can construct its page table so that the resolved physical addresses are the same as the starting virtual addresses. This allows the kernel to reside simultaneously at the same location in both physical and virtual memory; eliminating the circularity of page tables being kept in virtual memory. We believe that this is the most significant feature in a pedagogically undergraduate-appropriate hardware architecture. Undergraduate OS authors can now implement virtual memory based on the conceptual framework presented in their textbooks[19, 21, 22] (which tend to at best just wave their hand at the circularity issues described above), and have an easier time both with virtual memory debugging and for interacting with external block-oriented DMA devices (i.e. disks and tapes) that only understand physical addresses.

Outside of the simplification of the virtual memory management subsystem, μMPS is virtually identical to the MPS emulator upon which it is based. This includes:

- Support for up to eight memory-mapped DMA disk and tape devices.
- Support for up to eight memory-mapped printers and read/write capable terminal devices.
- Support for up to eight memory-mapped ethernet network devices. While the base Kaya OS project does not include supporting the network interface, it is described as an "additional" project.
- A sophisticated development, user interface, testing, and debugging environment.

It is important to observe that the μMPS hardware emulator including its user interface and its debugging environment is independent of the Kaya OS. Hence the μMPS emulator can be used not only with other OS designs, but as a hardware emulator in its own right like SimOS[20] and vmips[7]



```
prevPC: 0xFFFFFFFF : (nop)
    PC: 0x1FC00000 : BEQ $00, $00, +396
Stop:USER:(BOOTROM+0x0)
```

```
$00(00)  :  0x00000000          INDEX  :  0x00000000
$01(at)  :  0x00000000         RANDOM  :  0x00000E00
$02(v0)  :  0x00000000        ENTRYLO  :  0x00000000
$03(v1)  :  0x00000000       BADVADDR  :  0x00000000
$04(a0)  :  0x00000000        ENTRYHI  :  0x00000000
$05(a1)  :  0x00000000         STATUS  :  0x10400000
$06(a2)  :  0x00000000          CAUSE  :  0x00000000
$07(a3)  :  0x00000000            EPC  :  0x00000000
$08(t0)  :  0x00000000           PRID  :  0x00000166
$09(t1)  :  0x00000000         nextPC  :  0x1FC00004
$10(t2)  :  0x00000000         succPC  :  0x1FC00008
$11(t3)  :  0x00000000          ToDHI  :  0x00000000
$12(t4)  :  0x00000000          ToDLO  :  0x00000000
$13(t5)  :  0x00000000          Timer  :  0xFFFFFFFF
$14(t6)  :  0x00000000     prevPhysPC  :  0xFFFFFFFF
$15(t7)  :  0x00000000     currPhysPC  :  0x1FC00000
$16(s0)  :  0x00000000
```

Figure 1: The μMPS Main Window

3. THE KAYA OPERATING SYSTEM

Even if one starts with a high quality hardware emulator, like μMPS, the complexity of not only designing a student OS, but in laying the task out in an orderly, layered, multi-assignment manner is such that most instructors will not attempt it. The curricular materials that accompany μMPS presents an attempt at a "turn key" system for instructors. Most importantly this includes the specification for an OS, the Kaya OS –which is based heavily on the the Tina OS specification– whose implementation is broken down into multiple discrete project layers/assignments.

The curricular materials take the form of a Student Guide and an Instructor's Guide. The Student Guide details the μMPS architecture, its GUI interface, a debugging guide, a detailed specification of the Kaya OS, and the five layers or assignment phases that gradually build Kaya. The Instructor's Guide in addition to containing a complete well documented sample solution, guides instructors through the installation process, and presents many helpful hints for those utilizing the project.

Specifically the Kaya OS is defined by:

- Layer/Phase 1: Based on the key OS concept that active entities at one layer are just data structures at lower layers, this layer is a data structure assignment. Students write code to manage queues of structures (i.e. process blocks). Some of these queues will eventually represent the ready queues, others will represent queues associated with semaphores, and one of these queues will be the queue storing the unused process blocks themselves. This layer also contains the data structure to manage a list of "active" semaphores.

- Layer/Phase 2: The Nucleus. This layer, which is built on top of layer 1, contains the scheduler and a simple deadlock detection mechanism. Low level process management and synchronization primitives are also part of this level. This includes process creation, deletion, semaphore wait and signal, and timing mechanisms. Finally, this layer implements the basic lower level device drivers; interrupt handlers for the supported I/O and clock devices.

- Layer/Phase 3: The VM-I/O Support Layer. This layer, which is built on top of layer 2, embodies the concept of multiple user-level cooperating processes that can request I/O and which run in their own virtual memory space. In addition to supporting virtual memory, this layer implements user-level synchronization primitives, a process delay facility, and the basic upper level device drivers for initiating I/O operations.

  Upon the completion of this level students will have an OS capable of running up to eight simultaneous user programs. These user programs for which we strongly encourage the students to create for themselves can read/write the terminal, write to a printer, cooperate with other user processes, in addition to reading/writing the disk devices. The excitement and sense of accomplishment that a student gets when she runs her own user programs (written in C) on her own OS, comprised 100% of her own code, is simply amazing.

- Layer/Phase 4: The File System. This layer which is built on top of layer 3 implements the abstraction of a flat file system. This layer contains the primitives necessary to create, rename, delete, open, close, and modify files.

- Layer/Phase 5: The Network Interface. This layer, which is also built on top of layer 3, implements a network protocol stack to utilize the ethernet network interface. Because the network interface is based on VDE[4], one Kaya user process can communicate with another Kaya user process running on another instance of their OS on either the same host or a different host. Furthermore, since VDE runs on most UNIX'es, a Kaya user process can also communicate with a standard UNIX process running on the same or different host.

While there are five assignments, past experience with CHIP indicates that the first three sufficiently occupy a fourteen week semester. After completing phase 3, students have a complete though unsophisticated OS that can execute their own student written programs. Phases 4 and 5 are included for purposes of instructor flexibility; using all five assignments over a two quarter course sequence or giving the students the first phase and having the students write their own phases 2, 3, and 4 or 5. Interestingly, while the first assignment is not OS specific, it does allow for students to become accustomed to the programming environment and the μMPS GUI in a gentle manner that does not require the quick mastery of newly presented OS concepts.

In addition to the Kaya OS project, which is an adaptation of the Tina OS specification for μMPS, work is currently underway to adapt the message-passing based AMIKE OS specification for μMPS as well.

4. RELATED OS COURSEWARE SYSTEMS

Almost all of the other current OS courseware systems are designed to have students write or replace major parts of an OS. Hence none share the philosophy behind μMPS/Kaya: start only with a hardware emulator (i.e. no initial OS code base for students to build on/replace) and end with a completely student written OS capable of running student written C programs. These alternatives to μMPS/Kaya provide students with a complete or partially complete OS. Students, through the course of the term, not only study the (often copious) supplied code base, but replace select modules with ones of their own construction and/or add completely new modules.

More specifically:

- Minix[23]: While not a hardware emulator nor really a courseware system per se, it is included for the sake of completeness. The philosophy behind Minix is not to have students write their own OS through module replacement, but to study and play with a production-like OS running on real hardware.

- OSP[12, 13] and its recent update to Java, OSP2[14] are integrated OS simulators. The student is presented with a complete system which includes a hardware architecture, a complete OS and a user configurable set of processes that run on the system. Students substitute provided modules for ones of their own, remake the system and run the simulation. The simulation not only simulates the running of the OS on the simulated architecture, but also the performance of the OS under a simulated load.

  In OSP2, since the implementation language is Java the simulated machine is seen as a set of Java classes. Unfortunately, this implies that all requests to/for hardware functionality are managed as standard method calls. Also there is no user interaction during execution, nor an integrated debugger. All user feedback is in the form of copious runtime statistics generated by OSP during the simulated execution of the predefined user tasks. Finally, given OSP's focus on performance, this system seems best suited for when the curricular goals are not primarily understanding operating systems from a "gestalt" perspective, but on the relative performance of one algorithm (e.g. scheduling or page replacement) over another.

- Nachos[3] is another integrated OS simulator. Much has been written regarding the shortcomings of Nachos[9, 5]. In summary it will suffice to observe that the OS is compiled together with the simulator and ran as a single executable. User processes are then ran in this simulator. This "mixed-mode" operation causes many conceptual and debugging problems (e.g. dealing with Nachos user programs in little-endian mode running on an OS/hardware simulator running in big-endian mode) in addition to blurring the hardware-software interface. Furthermore, all hardware artifacts are seen/accessed only through class interfaces.

- Topsy[8] is a micro-kernel OS written to run on a MIPS emulator or on real sparc or i386 hardware. Topsy is not a complete OS. It lacks, among other features, support for paged virtual memory or file system operations. While this leaves room for students to both enhance and expand Topsy, no curricular materials are provided to guide the process. Furthermore there is only limited support for emulated external devices.

- OS/161[9], like Topsy, is a partial OS designed to run on a MIPS-based hardware emulator (System/161). Students are left with six primary tasks which include rewriting the scheduler, implementing a full-featured virtual memory system and a file system. Unfortunately, System/161, like Nachos and Topsy, faithfully emulates the MIPS R3000 architecture. As observed above, real commercial machine architectures in general and the MIPS R3000 in particular, especially with respect to virtual memory management, are overly complex and unnecessarily complicate students' understanding. In particular, OS/161 by necessity contains an initial VM module so as to prevent requiring the VM module as the first assignment; virtual memory is always on in the MIPS R3000 architecture. The initial supplied VM module only allows one process to run at a time, thus requiring the student OS authors to first handle virtual memory management prior to the conceptually easier tasks of scheduling and multiprocessing.

  In many respects OS/161–System/161 seems designed to address the primary shortcomings of Nachos, eliminating the "mixed-mode" execution and the presentation of the hardware through C++ classes, while preserving a similar assignment structure.

- GeekOS[10] is a tiny OS written to run on an emulated i486 architecture. Unlike OS/161 or Topsy, the supplied kernel is not meant to be enhanced. Instead it acts like an extension to the hardware creating a virtual machine for which student operating systems are to be written. While this virtual machine approach allows one to hide/abstract away many of the complexities of working with a real commercial architecture, too many aspects are hidden/abstracted away. This means that students cannot get exposed to and therefore cannot deeply experiment with/understand real kernel–hardware interactions. Nevertheless, GeekOS is probably closest in spirit to the philosophy of μMPS/Kaya.

The desired components for an OS courseware system should include a sophisticated user interface and debugging facility, the ability to directly interact with the (emulated) hardware, a clean perception of the hardware/software interface (i.e. do not hide access to a hardware artifact behind a software interface), an implementation language that does not hide memory management issues (i.e. Java), little or no supplied OS code, and appropriate accompanying curricular materials of superior quality. Each of the above courseware systems has some of these components (Minix of course being excused for the amount of supplied code), though none have all of them. In fact it is observed that none possess sophisticated user interface and debugging environments; all are limited to command-line interaction which at best use gdb for debugging.

Finally, one might infer from the above list that for undergraduates, writing a complete though unsophisticated OS for a real commercial machine architecture is too complex, and that this complexity can be reduced to a manageable level by providing an initial OS code base and in some cases mitigating direct interaction with the emulated hardware through either Java classes or a supplied kernel layer. The μMPS/Kaya approach differs from this by asserting that writing a complete though unsophisticated OS for a pedagogically appropriate machine architecture, instead of a real commercial machine architecture, is not only a reachable goal for undergraduates, but one better fills the objectives of the undergraduate OS course project.

5. CONCLUSIONS

The μMPS/Kaya courseware system represents both a continuation of an established pedagogic approach to teaching the OS course as well as an alternative to the current crop of OS courseware systems. From a Bloom's taxonomy perspective, between supplying code for students to read and understand (Minix), supplying code for students to read, understand, modify and possible extend (OSP, Nachos, OS/161, Topsy, and GeekOS), and supplying no code and having students write all the code from scratch themselves, the latter approach arguably achieves the desired levels of analysis and synthesis.

Furthermore, using an integrated OS simulator or modifying discrete components of an already existing OS does not provide the same pedagogic opportunities/advantages that exist when developing a complete, though unsophisticated, OS. Consider the following scenario dealing with CPU scheduling: When using an OS simulator or modifying an existing OS students typically replace an unsophisticated round-robin CPU scheduling module with a module implementing the more sophisticated multiple level feedback queues algorithm. When using the μMPS/Kaya approach students typically only implement the unsophisticated round-robin scheduling algorithm. However, since they also write the code for all the modules that interact with the scheduler, students also see how the scheduler interacts with deadlock detection, the interrupt handlers, semaphore processing, and time accounting issues. In the first case students miss experiencing the complexity and concurrency that exists between OS components; some of the very attributes that make operating systems interesting to study. Alternatively, students who write their own OS can easily extrapolate the implications of substituting a more sophisticated algorithm for their round-robin scheduler.

OS courses do not persist in our curricula to prepare students for careers in OS authorship. Yet most of the existing OS courseware systems by only allowing students the opportunity to replace supplied simplistic modules with more sophisticated ones focus primarily on OS performance issues. Operating systems are studied because they embody algorithmic problem solving and the taming of complexity through concurrency and abstraction. μMPS, unlike existing courseware systems, more directly supports this goal due to its focus on the OS as a whole.

While it is universally believed by students that implementing a complete OS is not for the faint of heart, it is only commonly held that running a course where students implement a complete OS, regardless of the pedagogical value achieved, is not for the faint of heart. Fortunately, neither is true.

6. ACKNOWLEDGMENTS

We wish to acknowledge the support given to Michael Goldweber from Xavier University via a Faculty Development Grant and the NSF via a CCLI-EMD grant (#0341003), and the support given to Renzo Davoli by the WebMinds *FIRB* Project of the Italian Ministry for University Research and Education.

7. REFERENCES

[1] O. Babaoglu, M. Bussan, R. Drummond, and F. Schneider. Documentation for the CHIP computer system, 1988.

[2] O. Babaoglu and F. Schneider. The HOCA operating system specifications, 1990.

[3] W. A. Christopher, S. J. Procter, and T. E. Anderson. The nachos instructional operating system. In *USENIX Winter 1993 Conference Proceedings*, 1993.

[4] R. Davoli. VDE: Virtual distributed ethernet. In *Proceedings of Tridentcom*, 2005.

[5] R. Davoli and M. Goldweber. New directions in operating systems courses using hardware simulators. In *Proceedings of the SCS International Conference on Simulation and Multimedia in Engineering Education*, 2003.

[6] E. Dijkstra. The structure of the THE multiprogramming system. *Commun. ACM*, 11(3), may 1968.

[7] B. Gaeke. The vmips project. http//www.dgate.org/vmips.

[8] G.Frankhauser, C. Conrad, E. Zitler, and B. Plattner. Topsy - a teachable operating system. http://www.tik.ee.ethz.ch/∼topsy.

[9] D. A. Holland, A. T. Lim, and M. I. Seltzer. A new instructional operating system. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, 2002.

[10] D. Hovemeyer, J. Hollingsworth, and B. Bhattacharjee. Running on the bare metal with GeekOS. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, 2004.

[11] Joint IEEE Computer Society/ACM Task Force on Computing Curricula. Computing curricula 2001: Computer science - final report, 2001.

[12] M. Kifer and S. Smolka. *OSP An Environment for Operating System Projects*. Addison–Wesley, 1991.

[13] M. Kifer and S. Smolka. *OSP An Environment for Operating System Projects: Instructor's Manual*. Addison–Wesley, 1991.

[14] M. Kifer and S. Smolkaka. OSP 2. http://www.lmc.cs.sunysb.edu:8180/osp2/index.jsp.

[15] M. Morsiani. AMIKE resource page. http://www.cs.unibo.it/∼morsiani/amike.html.

[16] M. Morsiani. ICARO.S resource page. http://www.cs.unibo.it/∼morsiani/icaros.html.

[17] M. Morsiani. MPS resource page. http://MPS.sourceforge.net.

[18] M. Morsiani and R. Davoli. Learning operating systems structure and implementation through the MPS computer system simulator. In *Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education*, 1999.

[19] G. Nutt. *Operating Systems: A Modern Perspective*. Addison Wesley, 3rd edition, 2004.

[20] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer simulation: The SimOS approach. *IEEE Parallel and Distributed Technology*, 1995.

[21] A. Silberschatz, P. Galvin, and G. Gagne. *Operating System Concepts*. Wiley, 6th edition, 2001.

[22] A. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2nd edition, 2001.

[23] A. Tanenbaum and A. Woodhull. *Operating Systems: Design and Implementation*. Prentice Hall, 2nd edition, 1997.