The Pandos Project and the μMPS3 Emulator



*Published Version:*
Goldweber, M., Davoli, R., Biondi, M. (2021). The Pandos Project and the μMPS3 Emulator. New York : ACM [10.1145/3430665.3456331].

*Availability:*
This version is available at: https://hdl.handle.net/11585/826984 since: 2021-06-28

*Published:*
DOI: http://doi.org/10.1145/3430665.3456331

# The Pandos Project and the $\mu$MPS3 Emulator

*Michael Goldweber* — *Xavier University* — *mikeyg@xavier.edu*
*Renzo Davoli* — *Università di Bologna* — *renzo@cs.unibo.it*
*Mattia Biondi* — *Università di Bologna* — *mattiabiondi1@gmail.com*

**Abstract**

This paper introduces $\mu$MPS3, a new system emulator based on the MIPS (R3000) architecture. We characterize $\mu$MPS3 as a cross between a real MIPS system and the typical RISC machine architecture presented to students in operating systems textbooks; a realistic, but not excessively complex architecture. Furthermore, we present Pandos, a specification for a one semester, multi-phase project for the development of a complete operating system for deployment on $\mu$MPS3.

Alternative operating systems projects (e.g. NachosChristopher et al. (1993)) provide students with a significant starting code base. Students then modify existing modules or add new ones. Our new courseware system is predicated on the observation that learning outcomes for artifact-based courses are best achieved by having students create an instance themselves. With $\mu$MPS3/Pandos students undergo the pedagogically valuable experience of starting only with a system emulator and ending with a completely student written operating system capable of running student written C programs.

$\mu$MPS3 is a significant update of $\mu$MPS, originally released over fifteen years ago. ($\mu$MPS2 upgraded $\mu$MPS with multiprocessor support.) In addition to updating $\mu$MPS3 to easily install on student laptops, $\mu$MPS3/Pandos endeavors to address the key conceptual stumbling blocks students faced when writing a complete operating system for these earlier systems. The successful conclusion of the "core," one semester Pandos project results in a working operating system supporting a variety of different peripheral device classes, process creation/termination, concurrency primitives, and up to eight concurrently executing user processes, each running in their own logical address space.

## 1 Introduction

While undergraduate computing curricula have evolved over the past forty plus years, an operating systems course has remained a consistent fixtureon Computing Curricula: Association for Computing Machinery (ACM) and Society (2013); McGuffee (2020). This is in spite of the observation that the vast majority of our undergraduates will never be professionally engaged in the writing or maintenance of an operating system. What are the learning outcomes from an operating systems (OS) course that are so enduring?

Mastering topics such as the process/thread model and concurrency are central not only to operating systems, but to algorithmics in general, hence one might find these topics in other courses (e.g. database systems). One unique way the operating systems course stands out is that it affords students the opportunity to wrestle with and master complexity to a depth not otherwise available in the undergraduate computing curriculum. This is particularly true with respect to how hardware interacts with software (exception and interrupt handling), concurrency, CPU and device scheduling, and virtual memory (e.g. TLB refill and page replacement algorithms).

An operating system is a software artifact. Like all software artifact-based courses (e.g. compilers, neural networks), the gold standard accompanying lab exercise is to build an instance of the artifact. Theoretical explanations and small-to-medium size programming projects notwithstanding, they pale to the pedagogic value of having students read, understand and modify code from a large, complex system, or better yet, as we assert above, write their own operating system, maybe entirely from scratch.

The discussion on how to best accomplish this has been ongoing for many yearsLions (1977); Yun-Lin (1989); O'Gorman (1998); Sheehan (2007); Dall and Nieh (2014); Buck and Perugini (2019b). Perspectives are typically characterized along two orthogonal axes:

- Platform
  - (A) Working with real hardware.
  - (B) Working with emulated hardware.
- System
  - (1) Working with a production quality operating system.
  - (2) Working with an education-oriented operating system.

Choice B-2 can be further broken down to two subcategories: education-oriented operating systems that run on emulated hardware, or systems that emulate both the hardware and the operating system as a single unit. Hence student activities can vary from working with complex, real operating systems, to educational, but still realistic ones, to studying/modifying existing software, to writing new creations from scratch.

Further complicating this landscape are the following two observations:

- The operating system course, and accompanying lab project, is very often limited to a single semester.
- The final artifact must transcend the "toy" threshold. While there is no set definition for what makes a student modified/written operating system "real" we posit that the final artifact should operate as one would expect: successfully run multiple concurrently executing programs. Preferably, these programs should be student written.

In spite of all the different possible choices, working with an education-oriented operating system in conjunction with emulated hardware is seemingly the most popular. The reason for this boils down, primarily, to complexity/cognitive load. Real hardware, including RISC-based machines, are still, nonetheless, very complex, as are the production operating systems created to run on them. With only a semester to work with, the myriad of details to master, with respect to real hardware, becomes insurmountable. This is true whether one is writing a simple operating system for real hardware, or modifying a production operating system.

The Virtual Square Lab has been researching, developing, and releasing hardware emulators to support the operating systems course. $\mu$MPS3, a new hardware emulator based on the MIPS (R3000) architecture, is the culmination of over twenty years of working in this space. We characterize $\mu$MPS3 as a cross between a real MIPS based system and a typical machine architecture presented to students in operating systems and machine organization textbooks; a realistic, but not excessively complex architecture.

Pandos is the education-oriented operating system that students create to run on $\mu$MPS3. Pandos is not a downloadable software artifact, but a multi-phased design of an operating system. The core Pandos project is a semester-long 3-phase project at whose successful conclusion results in a working operating system supporting a variety of different peripheral device classes (printers, disks, terminals, and flash devices), process creation/termination, concurrency primitives, and up to eight concurrently executing user processes (C programs), each running in their own logical address space.

Since $\mu$MPS3 is a separate MIPS hardware emulator, it is capable of running any compatible software, of which Pandos is but one example. Similarly, Pandos interacts with the hardware in an authentic manner. If one were to build actual hardware to the $\mu$MPS3 specification, any student created Pandos would run unaltered.

The pedagogic philosophy behind $\mu$MPS3/Pandos is simplicity coupled with completeness. Under Pandos, students implement a very simple scheduler (round robin), a simple deadlock detection algorithm and a simple page replacement algorithm (FIFO). The complexity arises from how each part of the operating system interacts with each other part.

The best way to learn about operating systems is to have students write a complete, realistic, but not production quality, operating system from scratch. Pandos is an operating system design to facilitate student authorship of a “complete” operating, with no initially provided code, to run on “hardware” designed not for super fast instruction execution (i.e. a real RISC/CISC machine), but to be both realistic and accessible to undergraduate students – $\mu$MPS3.

The rest of this paper is organized as follows. Section 2 presents a concise evaluation of related courseware systems. The $\mu$MPS3 emulator is described in Section 3, while Pandos is described in Section 4. Finally, the paper concludes with Section 5.

## 2 Related Projects

As described by Otero and AravindRomán Otero and Aravind (2015) there are five basic ways to support the operating systems course:

- No programming projects at all. (Limited learning experience.)
- Examine various aspects via dedicated simulatorsHoskey (2013); Miley and Leverington (2016); Buck and Perugini (2019b,a) or standalone programming projects. (Capture higher level functionality, but are unmoored from real implementations.)
- Study/Modify a commercial operating systemLions (1978); O'Brien (2017). (Huge learning curve limits students to very narrow views of a small, select set of modules.)
- Modify/extend an instructional operating system.
- Build a (hopefully non-toy) operating system from scratch.

Furthermore, we echo Otero and Aravind's criticisms of the first three approaches, summarized as parenthetical comments above.

This leaves:

### 2.1 Modify/extend an instructional operating system

In this approach, students are provided either a partial or complete code base. Those with a partial code base (e.g. TopsyFankhauser et al. (2000), OS/161Holland et al. (2002), Awk-LinuxCheng and Lin (2008), GeekOSHovemeyer et al. (2004)) require students to complete the kernel and extend it with additional modules. How much code, and which modules are provided vary from system to system. We still see this pedagogically as a breadth versus depth question. The key learning outcomes for the operating system lab experience are best served by an approach with the greatest breadth.

Those with a complete code base (e.g. OSP/2Kifer and Smolka (1992, 2007), MINIXTanenbaum (2006), PintOSPfaff et al. (2009), BabyOSLiu et al. (2007), Xv6Cox et al. (2020)[1]) follow the recommendation that the best way to learn operating systems is by studying the code from a complete operating systemLions (1978); either educational or production. Students, not only study the code base, but enhance it by writing replacement modules, and/or extend it with new modules of their own. However, students miss out on understanding complex module interactions when they simply upgrade a given existing module's algorithm. (e.g. Replace the round robin scheduler with a more sophisticated one.)

Some of these systems integrate the operating system with the emulator (i.e. the emulator and the operating system are compiled together into one executable) blurring the hardware-software interface (e.g. OSP). The superior approach is to have a separate emulator which then runs an independently compiled operating system. (e.g. System/161 & OS/161.) These more complex environments however require correctly configured cross compilers.

These systems do resemble real operating systems, however they often hide actual hardware-software interaction and the complexity that comes from module interaction. Reading code, however valuable, is not the same as writing code. Similarly, updating and extending a large code base, while also a valuable

[1] Xv6 was recently updated from the PDP-11 architecture to RISC-V.

experience is not the same as building a complete instance of the artifact oneself. Upgrading a module (e.g. scheduler) still hides, to a large degree, how all the pieces parts fit together.

## 2.2 Build a (hopefully non-toy) operating system from scratch

These more complex systems require a framework for students to build their own complete operating system. As described in Section 3, one of the earliest is the CHIP/HOCA systemBabaoglu and Schneider (1983), which we consider to be the intellectual ancestor of $\mu$MPS3/Pandos. PortOSAtkin and Sirer (2002) is another outdated and seemingly no longer supported example.

Unlike CHIP, which is an emulator for a hypothetical architecture, NachosChristopher et al. (1993) is based on the MIPS architecture. As with some of the above systems, Nachos integrates the operating system with the simulator, introducing conceptual problems revolving around the distinction between hardware and software. Since Nachos is an OO-based system (C++ or Java), hardware artifacts are presented as class objects. The Nachos project is so large and complex, it rivals small production operating systems in its complexity. Multiple supplemental guides have been produced to aid studentsNarten (1997); Hettena and Cox (2000). Furthermore, many adopters provide significant amounts of starting code, eliminating its “build the whole system” premise. The PintOSPfaff et al. (2009) project, which runs on the x86 ISA, real or emulated, is an attempt to address Nachos's shortcomings.

Given the problem that real architectures/hardware are overly complex, and the tricky balancing act that systems such as $\mu$MPS3 try to achieve –real architectures simplified just enough for student success– a third approach has seen recent interest; use real, but simplified hardware; SoC boards.

- The Incredibly Simple Operating SystemMertz (2017). Using a Zybo board, students create a simple shell environment and routines to support a small set of API calls.
- vmwOSFrancis-Mezger and Weaver (2018). Using a Raspberry Pi, students create a simple operating system that supports multitasking. While virtual memory is addressed, the page tables are all 1:1 and it is not used for inter-process protection.
- MiniOSRomán Otero and Aravind (2015). Using the ARM-based Atmel SAM4S Xplained Pro, students create an operating system capable of interacting with native peripheral devices (though the device drivers are native), multitasking, and a file system.

While intriguing, the difficulty of the SoC board approach is that their architectures do not resemble those described in mainstream operating system textbooks; especially with respect to address translation/memory management. Hence the resulting “embedded” operating systems do not resemble the operating systems they are familiar with. Nor will they behave in the expected manner; running multiple student-written C programs, each memory protected from each other, utilizing terminal I/O. Finally, while playing with real hardware presents a unique “fun factor,” the test - modify - retest cycle is considerably more cumbersome.

While this brief summary demonstrates the domain's richness, many of these systems are either no longer supported/available (e.g. BabyOSLiu et al. (2007), Black's projectBlack (2009), GeekOSHovemeyer et al. (2004), OSP/2Kifer and Smolka (1992, 2007), PortOSAtkin and Sirer (2002), TopsyFankhauser et al. (2000)), or lack sufficient documentation for use beyond their developers (e.g. Awk-LinuxCheng and Lin (2008), ISOSMertz (2017)). The last (informal) survey of usage of these systemsAnderson and Nguyen (2005) is too old to provide meaningful data. However, we surmise that the still supported/available systems that see the most use are also those with sufficient documentation (e.g. NachosChristopher et al. (1993); Hettena and Cox (2000), OS/161Holland et al. (2002), PintOSPfaff et al. (2009)) to allow adopters from other institutions to be successful.

## 3 The $\mu$MPS3 Emulator

While the full history of hardware emulators and accompanying operating system projects designed for students to write a complete, though simple, operating system from scratch is beyond the scope of this paper, we acknowledge the Cornell Hypothetical Instruction Processor (CHIP) and its accompanying operating systems project, HOCABabaoglu and Schneider (1983), as one of the early success stories. Inspired by CHIP/HOCA, the Virtual Square Lab, in 1999, released MPSMorsiani and Davoli (1999), a machine emulator that authentically emulated a MIPS 3000 processor, without floating point, in addition to supporting various device categories.[2] The simulator was accompanied by the specification for an operating system project: TINA - a HOCA inspired project designed for MPS. Though MPS proved useful for various operating systems research projects, it was, after all, an authentic emulator with an excellent GUI-based debugger, it was not successful in the classroom. Writing an operating system for a "real" MIPS 3000 was simply too complicated for undergraduates in a single semester.

The lessons of MPS led to the Virtual Square Lab's development of $\mu$MPS and its accompanying TINA inspired project; KayaGoldweber et al. (2005). $\mu$MPS "simplified" the MIPS R3000 architecture by

- Adding a VM bit.
- Forcing a fixed page table and segment table format so that the BIOS routines handled all TLB refill events.

$\mu$MPS/Kaya was quite popular in its day, though it was a number of years before it was learned that many adopters opted to forego a full implementation of the final phase; support for virtual memory. In 2012, $\mu$MPS was given multiprocessor support ($\mu$MPS2)Goldweber et al. (2012). Since Kaya is a uniprocessor project, there was no appreciable change to the project.

In an attempt to address both the complexity of Kaya's final phase (address translation) and the observation that the MIPS R3000 architecture had become rather out of date, the Virtual Square Lab produced $\mu$ARM and its accompanying port of Kaya, relabeled JaeOSMelletti et al. (2015). The $\mu$ARM emulator is to an ARM7tdmi-based system what the $\mu$MPS2 emulator is to a MIPS R3000-based system; a pedagogically simplified CPU embedded in an easy to understand system architecture.

$\mu$ARM/JaeOS was inspired by a predicted migration in the assembly/machine organization prerequisite course from MIPS to ARM. As discussed below, this migration has yet to occur. However, over the twenty plus year span reaching back to the development of MPS, the Virtual Square Lab has learned many lessons regarding the appropriate mix of architectural authenticity and pedagogically inspired simplicity. These lessons, enumerated below drove the design of $\mu$MPS3

- Segmentation adds unnecessary complexity. Abandoning segmentation and the concomitant segment tables not only simplifies the student software, but is in keeping with current architectural trends.
- Architectural simplicity must take precedence over popularity. The ARM (7) architecture, with its seven operating modes, differing sets of banked registers (dependent on operating mode) and nonuniform exception handling is simply too complex for student success in a one semester time frame; especially when compared against the MIPS architecture. Furthermore, given the rising popularity of RISC-V and the tremendous similarity between MIPS and RISC-V, the MIPS architecture currently feels less "out of date" than it did a decade ago.
- In a similar vein, we've learned from adopters that the choice of operating system project is highly dependent on the choice of assembly language and related architecture covered in the prerequisite machine organization/assembly course. MIPS remains one of the two most popular choices, along with the x32/64, for teaching assembly language programming. One probable reason for this is the mature set of student-

[2] As a sign of how times have changed, mag tape was one of MPS's device categories.

oriented MIPS programming environmentsLarus (2018); Vollmar and Sanderson (2006). No such set of mature tools exist (yet) for either ARM or RISC-V.

- A single exception/interrupt entry point, which therefore requires a *Cause* register, is evidently easier for students to work with than a vectored exception/interrupt approach. This is most likely due to fewer regions for student code to initialize. It does require the seemingly simpler task of examining the *Cause* register prior to a multi-way branch.

- BIOS code which does less is evidently more. TLB-based address translation mechanisms generate two types of exceptions: TLB-refill events and page faults. Hiding TLB-refill processing in the BIOS subsystem did not accomplish the intended goal of aiding students. TLB-refill events would still occur and be seen while debugging, but remained confusingly opaque.

- The key conceptual hurdle when implementing/supporting virtual memory is the problem of having kernel code that can access logical addresses and physical addresses simultaneously. MPS treated all addresses as logical. $\mu$MPS & $\mu$MPS2 implemented a VM bit, but as under MPS, still required specifically crafted 1:1 page tables to solve this. Taking a cue from $\mu$ARM, which reserved the first eight frames of RAM as exempt from address translation, $\mu$MPS3 implements a boot time configurable translation floor address, below which address translation is disabled. Hence, $\mu$MPS3 can behave like MPS (address translation always on), or have translation enabled for processes' logical address space(s) and disabled for operating system code and data structures.

- The final significant hurdle was user process initialization; particularly the initialization of each process's backing store. With $\mu$MPS3's introduction of configurable flash devices, user processes can now be presented to the operating system as preinitialized on a backing store device.

Hence, $\mu$MPS3 is not backward compatible with $\mu$MPS2:

- The VM bit was removed and a translation floor address was introduced.

- The BIOS code was rewritten to remove TLB-refill functionality.

- Segmentation was removed.

- Flash devices were introduced along with a utility which allows for a preloading of an executable image. (i.e. backing store)

- The $\mu$MPS3 emulator and development toolchain (MIPS cross compiler) was configured for easy student installation. Previous Virtual Square Lab emulators and development toolchains were designed to be installed on a department server. (i.e. It was complicated.) Since students now prefer to work on their own machines, installation on any Debian/Ubuntu system has been reduced to a single package manager command.[3]

Furthermore, $\mu$MPS3 retains all the popular features of $\mu$MPS2:

- An emulated multicore MIPS R3000 cpu embedded in a larger emulated system containing a system bus, and a host of connected peripheral devices which include both block and character devices, and fully functioning network cards. All of the peripheral devices use a simple direct memory mapped architecture with a uniform device register format. The $\mu$MPS3 ISA is that of the R3000; students use the standard GNU C compiler configured as a MIPS R3000 cross compiler for the development toolchain.

- A BIOS subsystem whose primary functionality is the extension of the R3000 ISA with a handful of routines which greatly assist in student operating system authorship. (e.g. Atomic loading

[3] $\mu$MPS3 has been tested on various Debian/Ubuntu setups including, Linux machines, dual boot machines, and a variety of virtualization tools: Qemu-KVM, Parallels, VMware, and VirtualBox.

and storing of a complete processor state, access to individual control registers, etc.) Furthermore, the BIOS code which is loaded at system boot time (i.e. not baked into $\mu$MPS3) is part of the $\mu$MPS3 distribution allowing for its replacement/modification as an advanced student project.

- A sophisticated debugging environment which includes breakpoints, memory tracing, memory suspect breakpoints, in addition to special operating system debugging features: break on interrupt, break on exception, and break on VM event. Finally, $\mu$MPS3 allows for an OS reload. Hence, breakpoints, traces, etc. are maintained between test runs which can involve code changes and OS recompilations.

- The system is fully described in an extensive Open Educational Resource (OER) reference document. The $\mu$MPS3 Principles of Operation is the complete guide to the $\mu$MPS3 architecture; its emulator, GUI, debugging facilities, development toolchain usage, and installationGoldweber and Davoli (2020a).

The $\mu$MPS3 interface is composed of several windows which let the user exercise the desired level of control over the simulation execution and output, while offering a series of high level debugging tools:

- Code and TLB modification breakpoints
- RAM inspector
- Processor and coprocessor registers matrix
- TLB contents inspector

## 4 Pandos

Pandos is not an operating system one downloads to run on $\mu$MPS3. Instead it is the specification of an operating systems project, designed to be completed by undergraduate students during a single traditional semester. The specification is delivered via a separate OER reference document, the Student Guide to the Pandos ProjectGoldweber and Davoli (2020b). This OER[4] is a detailed project specification divided into multiple phases. The first three phases represent the "core" project. Six additional phases are also presented for follow up/advanced projects; three of which are described in full detail.

Pandos is a multi-layer operating system, strongly based on the Kaya OS specificationGoldweber et al. (2005). Each layer serving both as a support level for the above layers and as a specific assignment phase, with increasing difficulty and abstraction. C is the supported implementation language, but any programming language with a compiler capable of outputting MIPS binaries could work as well. While a prerequisite course in machine organization (with MIPS assembly programming) is recommended, it is not required. Debugging Pandos does require an understanding of register usage convention; parameter passing and exception handling.

The core Pandos project is structured as follows:

Phase 1: This first assignment is a simple data structures task designed to build student confidence with the emulator, the programming language and the debugging environment. Students write code managing all the data structures used by the kernel nucleus (Phase 2): Process control blocks (pcb) and semaphore queues. This phase is not strictly related to the course subject as semaphores and pcbs are only passive data structures at this point.

Phase 2: The second phase represents the kernel nucleus, providing low level (i.e. protected) system calls, a scheduling facility (round robin), exception and interrupt handlers, plus a rudimentary deadlock detector. Eight low level system calls provide services to the higher layers. These include process creation & termination, a semaphore blocking/unblocking service (i.e. P & V), an I/O synchronization service, and primitives to manage two different timing mechanisms. This phase takes advantage of the Phase 1 facilities to implement the "ready" queue, the semaphore blocked process queues, and I/O synchronization.

[4] Both OER documents are available from the $\mu$MPS3 website: `virtualsquare.org/umps`

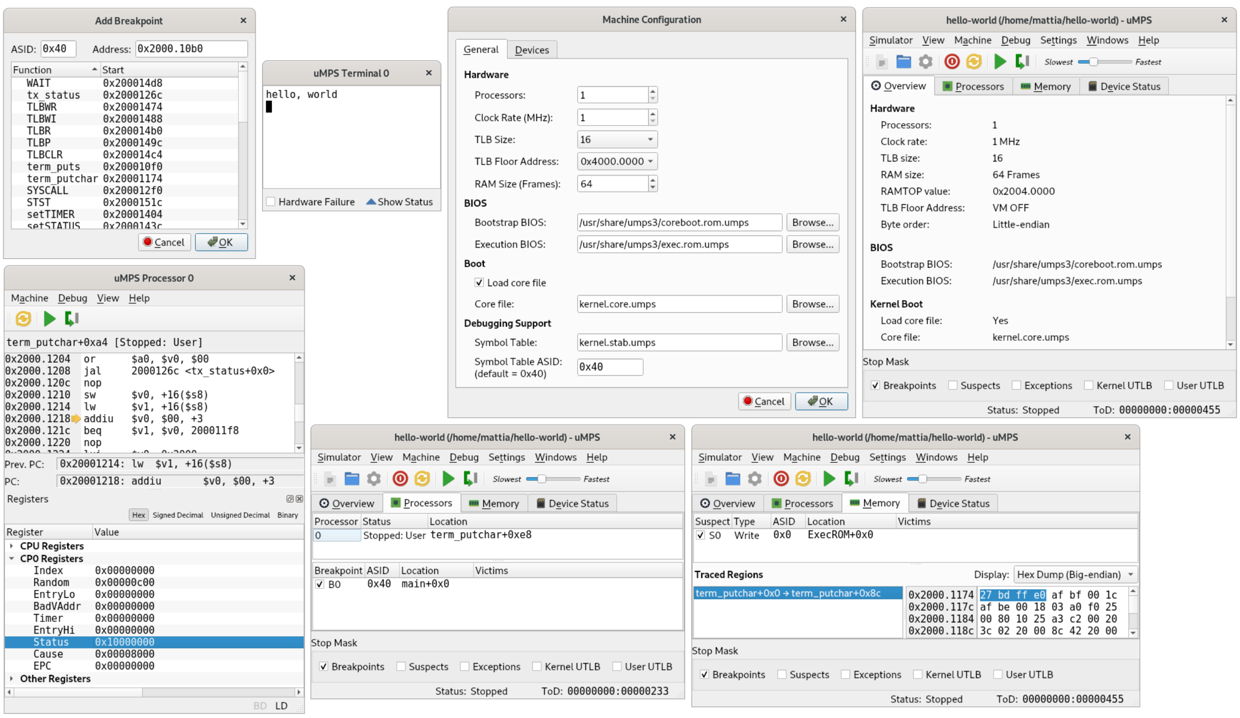


Fig. 1: Main GUI components

Phase 3: This layer provides an environment for up to eight user processes to concurrently execute in their distinct, but identically structured logical address space. Students implement page tables (simple arrays), the TLB-refill event handler, a pager (demand paging with a global FIFO page replacement alg.), and the high level system calls to support user process printer and terminal I/O.

At this point Pandos is capable of running multiple simultaneous interactive user programs. Since user programs are just C programs, we encourage students to write their own test/user programs.[5] Past student examples include an interactive palindrome checker, or recursively computing the $n$th fibonacci number, where $n$ is input by the user.

The Pandos Guide goes on to fully specify three advanced projects:

Phase 4: The high level system calls to support user process disk and flash device I/O.

Phase 5: A Delay Facility. This phase, illustrative of a simple daemon kernel thread, implements a service where user processes "sleep" for a specified number of seconds.

Phase 6: The high level system calls to support P & V operations on logically addressed semaphores. In addition to allowing user processes to now cooperate, this phase provides a small logical address space shared among all user processes.

Unlike the first three phases, each of which builds on the previous phase, Phases 4–6 are fully independent of each other. Instructors and/or students can therefore choose which ones to undertake and the order of implementation, if more than one.

Based on past experience with $\mu$MPS2, other advanced student projects include:

- A file system. Building on Phase 4, students can build a simple flat file system. Implemented system calls include file creation, deletion, renaming, open, close, read, and write.
- An interactive user "shell" program. This phase is somewhat open-ended. Simple shell programs can provide interaction with the file system. More sophisticated versions can extend to process creation and management.
- Support for the network card interface. The objective of this layer is building a network protocol stack in order to have multiple emulators communicating with each other. Moreover, since network devices are based on the Virtual Distributed Ethernet (VDE) toolDavoli (2005), $\mu$MPS3 network devices can communicate with any virtual or physical host equipped with VDE. Thus it is possible to test Pandos network stacks in either real or emulated network environments.
- Rewrite/extend the BIOS code, which is written in MIPS assembly, to provide different or extended functionality.

In initial class testing (seniors with extensive Java experience and only a little C/C++ experience, working in teams of two) a large majority (85%) successfully completed the core project (Phases 1–3). For comparison, at these same institutions, when using $\mu$MPS2/Kaya, either the Kaya equivalent to Phase 3 was not attempted, or when attempted only had about a 25% completion rate.

It is important to observe that $\mu$MPS3 is in no way tied to Pandos. Hence, students, as well as other OS researchers, are free to develop other operating systems to run on $\mu$MPS3. While not necessarily appropriate for undergraduates, this represents the ultimate OS project; to not just implement an operating system (e.g. Pandos), but to design and implement a new operating system for the $\mu$MPS3 architecture/system.

## 5 Observations & Conclusions

After twenty years and multiple attempts at finding the right mix of architecture (realistic, but not excessively complex) and operating system project design, it would be hubris to claim that $\mu$MPS3/Pandos is the perfect solution. However, assuming the development of major courseware systems to support

[5] Sample user programs are available from the $\mu$MPS3 website.

complex projects is evolutionary, we believe that $\mu$MPS3/Kaya is a major step forward.

Longitudinal analysis by the authors at their home institutions augmented by anecdotal feedback from adopters of the Virtual Square Lab emulators over the years has yielded an understanding of the key implementation concepts students have the greatest difficulties with: simultaneous access to both logical and physical addresses, and backing store initialization.

It has been well over a decade, exclusive of the recent SoC experiments, since a new courseware system/project was released, that has seen widespread adoption, whose goal was to facilitate student authorship of a complete operating system from scratch. Anecdotal evidence strongly suggests that many educators have given up on this grail and have turned to other approaches.[6] Hence, it was time for the Virtual Square Lab to apply the lessons learned from past releases to not just develop $\mu$MPS3/Pandos, but also accompany it with what we hope are straightforward and exhaustive OER reference documents.

Initial class testing at the authors' home institutions indicate $\mu$MPS3/Pandos has the potential to infuse new enthusiasm for operating system educators to again achieve the pedagogic holy grail for this artifact-driven course; have students successfully build, from scratch, an instance of the artifact themselves.

It is also worth observing that in addition to mastering key operating system concepts, there are many secondary learning outcomes associated with $\mu$MPS3/Pandos:

- It may very well become many students' capstone programming experience. Any undergraduate who has written a complete operating system and then concluded the experience by running a program they wrote on their operating system has reason to be proud of the accomplishment. Furthermore, such accomplishments are what students mention first when asked about the highlights of their undergraduate careers.

- An opportunity to apply concepts learned in a Software Engineering course. Pair programming in particular, if teams are utilized.[7]

- The importance of documentation. Pandos is simply too large for students to keep all the details in their head; especially when towards the end of the project they need to refer to code written early on.

- Team skills, if one opts to utilize teams.

We believe that the key learning outcomes of mastering complexity (i.e. the myriad of interactions between modules and layers), understanding the hardware–software interface, and learning how both to implement and utilize concurrency are best accomplished by having students build an entire kernel. $\mu$MPS3/Pandos is a new courseware system/project designed to achieve this goal.

Future developments of the project will include the development of additional OERs. Some of these resources (e.g. explanatory videos) will be targeted toward the student Pandos authors. Other resources will be targeted towards adopters who while tasked to teach operating systems, may lack the confidence/experience, in spite of the comprehensive OER materials already available, to undertake a $\mu$MPS3/Pandos adoption. In one sense, the development of $\mu$MPS/Pandos, a tidy, project in a box solution, was targeted for this very audience.

## References

Charles L. Anderson and Minh Nguyen. 2005. A Survey of Contemporary Instructional Operating Systems for Use in Undergraduate Courses. J. Comput. Sci. Coll. 21, 1 (Oct. 2005), 183–190.

Benjamin Atkin and Emin Gün Sirer. 2002. PortOS: An Educational Operating System for the Post-PC Environment. In Proceedings of the 33rd SIGCSE Technical Symposium on Computer

[6] Birds of a Feather and other in/formal gatherings of operating system educators at regional, national, and international computer science education conferencesMcGuffee (2020).

[7] One of the authors, who uses teams of two, actively encourages pair programming. It is not unusual that towards the end of the project most of the teams are employing pair programming after observing the successes of early adopters.

Science Education (SIGCSE '02). Association for Computing Machinery, New York, NY, USA, 116–120. https://doi.org/10.1145/563340.563384

Ozalp Babaoglu and Fred Schneider. 1983. The HOCA Operating System Specifications. Technical Report. Cornell University. Technical Report TR 83-585.

Michael D. Black. 2009. Build an Operating System from Scratch: A Project for an Introductory Operating Systems Course. In Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE '09). Association for Computing Machinery, New York, NY, USA, 448–452. https://doi.org/10.1145/1508865.1509022

Joshua W. Buck and Saverio Perugini. 2019a. An Interactive, Graphical CPU Scheduling Simulator for Teaching Operating Systems. J. Comput. Sci. Coll. 35, 5 (Oct. 2019), 78–90.

Joshua W. Buck and Saverio Perugini. 2019b. An Interactive, Graphical Simulator for Teaching Operating Systems. In Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19). Association for Computing Machinery, New York, NY, USA, 1290. https://doi.org/10.1145/3287324.3293756

Y. Cheng and J. M. Lin. 2008. Awk-Linux: A Lightweight Operating Systems Courseware. IEEE Transactions on Education 51, 4 (2008), 461–467. https://doi.org/10.1109/TE.2007.912571

W. A. Christopher, S. J. Procter, and T. E. Anderson. 1993. The Nachos Instructional Operating System. In USENIX Winter 1993 Conference Proceedings.

Russ Cox, Frans Kaashoek, and Robert Morris. 2020. Xv6: a simple, Unix-like teaching operating system. https://pdos.csail.mit.edu/6.828/2020/xv6.html.

Christoffer Dall and Jason Nieh. 2014. Teaching Operating Systems Using Code Review. In Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14). Association for Computing Machinery, New York, NY, USA, 549–554. https://doi.org/10.1145/2538862.2538894

R. Davoli. 2005. VDE: virtual distributed Ethernet. In First International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities. 213–220. https://doi.org/10.1109/TRIDNT.2005.38

C Fankhauser, E Zitzler, and B Plattner. 2000. Topsy — A Teachable Operating System. http://www.tik.ee.ethz.ch/ topsy.

Pascal Francis-Mezger and Vincent M. Weaver. 2018. A Raspberry Pi Operating System for Exploring Advanced Memory System Concepts. In Proceedings of the International Symposium on Memory Systems (MEMSYS '18). Association for Computing Machinery, New York, NY, USA, 354–364. https://doi.org/10.1145/3240302.3240311

Michael Goldweber and Renzo Davoli. 2020a. $\mu$MPS3 Principles of Operation. Lulu.com.

Michael Goldweber and Renzo Davoli. 2020b. Student Guide to the PANDOS Project. Lulu.com.

Michael Goldweber, Renzo Davoli, and Tomislav Jonjic. 2012. Supporting Operating Systems Projects using the $\mu$MPS2 Hardware Simulator. In Proceedings of the $17^{th}$ Annual Conference on Innovation and Technology in Computer Science Education.

Michael Goldweber, Renzeo Davoli, and Mauro Morsiani. 2005. The Kaya OS Project and the $\mu$MPS Hardware Simulator. In Proceedings of the $10^{th}$ Annual Conference on Innovation and Technology in Computer Science Education.

Dan Hettena and Rick Cox. 2000. A Guide to Nachos 5.0j. https://inst.eecs.berkeley.edu/ cs162/sp07/Nachos/walk/walk.html

David A. Holland, Ada T. Lim, and Margo I. Seltzer. 2002. A New Instructional Operating System.

In Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE '02). Association for Computing Machinery, New York, NY, USA, 111–115. https://doi.org/10.1145/563340.563383

Arthur Hoskey. 2013. Simple OS: A Component-Based Operating System Simulator in the Spirit of the Little Man. J. Comput. Sci. Coll. 28, 6 (June 2013), 117–124.

David Hovemeyer, Jeffrey K. Hollingsworth, and Bobby Bhattacharjee. 2004. Running on the Bare Metal with GeekOS. In Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04). Association for Computing Machinery, New York, NY, USA, 315–319. https://doi.org/10.1145/971300.971411

Michael Kifer and Scott Smolka. 2007. Introduction to Operating System Design and Implementation: The OSP 2 Approach (Undergraduate Topics in Computer Science). Springer-Verlag, Berlin, Heidelberg.

Michael Kifer and Scott A. Smolka. 1992. OSP: An Environment for Operating System Projects. SIGOPS Oper. Syst. Rev. 26, 4 (Oct. 1992), 98–100. https://doi.org/10.1145/142854.142879

James Larus. 2018. SPIM: A MIPS32 Simulator. http://spimsimulator.sourceforge.net/.

J. Lions. 1977. A Commentary on the Sixth Edition UNIX Operating System. https://warsus.github.io/lions-/.

J. Lions. 1978. An Operating System Case Study. SIGOPS Oper. Syst. Rev. 12, 3 (July 1978), 46–53. https://doi.org/10.1145/775396.775402

Haifeng Liu, Xianglan Chen, and Yuchang Gong. 2007. BabyOS: A Fresh Start. In Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '07). Association for Computing Machinery, New York, NY, USA, 566–570. https://doi.org/10.1145/1227310.1227499

James W. McGuffee. 2020. Why Teach Operating Systems? J. Comput. Sci. Coll. 35, 9 (April 2020), 44–51.

Marco Melletti, Michael Goldweber, and Renzo Davoli. 2015. The JaeOS Project and the $\mu$ARM Emulator. In Proceedings of the $20^{th}$ Annual Conference on Innovation and Technology in Computer Science Education.

Andrew Mertz. 2017. The Incredibly Simple Operating System. J. Comput. Sci. Coll. 32, 5 (May 2017), 45–46.

Cayler Miley and Michael Leverington. 2016. A Hands-on Approach to Operating Systems through Simulation Projects. J. Comput. Sci. Coll. 32, 2 (Dec. 2016), 28–33.

Mauro Morsiani and Renzo Davoli. 1999. Learning Operating Systems Structure and Implementation through the MPS Computer System Simulator. In Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education.

Thomas Narten. 1997. A Road Map Through Nachos. https://users.cs.duke.edu/ narten/110/nachos/main/main.html.

Darragh O'Brien. 2017. Teaching Operating Systems Concepts with SystemTap. In Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17). Association for Computing Machinery, New York, NY, USA, 335–340. https://doi.org/10.1145/3059009.3059045

John O'Gorman. 1998. Teaching Operating Systems. SIGCSE Bull. 30, 2 (June 1998), 61–63. https://doi.org/10.1145/292422.292444

Joint Task Force on Computing Curricula: Association for Computing Machinery (ACM) and IEEE Computer Society. 2013. Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science. ACM.

Ben Pfaff, Anthony Romano, and Godmar Back. 2009. The Pintos Instructional Operating System Kernel. In Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE '09). Association for Computing Machinery, New York, NY, USA, 453–457. https://doi.org/10.1145/1508865.1509023

Rafael Román Otero and Alex A. Aravind. 2015. MiniOS: An Instructional Platform for Teaching Operating Systems Projects. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15). Association for Computing Machinery, New York, NY, USA, 430–435. https://doi.org/10.1145/2676723.2677299

Robert J. Sheehan. 2007. Teaching Operating Systems with Ruby. In Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '07). Association for Computing Machinery, New York, NY, USA, 38–42. https://doi.org/10.1145/1268784.1268798

Andrew Tanenbaum. 2006. Operating Systems: Design and Implementation (3rd ed.). Prentice Hall.

Kenneth Vollmar and Pete Sanderson. 2006. MARS: An Education-Oriented MIPS Assembly Language Simulator. In Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '06). Association for Computing Machinery, New York, NY, USA, 239–243. https://doi.org/10.1145/1121341.1121415

Su Yun-Lin. 1989. On Teaching Operating Systems. SIGCSE Bull. 21, 3 (Sept. 1989), 11–14. https://doi.org/10.1145/70593.70596