MINIOS: AN INSTRUCTIONAL PLATFORM FOR TEACHING OPERATING SYSTEMS LABS

by

Rafael Román Otero

B.Eng, Universidad del Valle de Mexico, Puebla, Mexico, 2009

THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER SCIENCE

UNIVERSITY OF NORTHERN BRITISH COLUMBIA

July 2016

Abstract

Delivering hands-on practice laboratories for introductory courses on operating systems is a difficult task. One of the main sources of the difficulty is the sheer size and complexity of the operating systems software. Consequently, some of the solutions adopted in the literature to teach operating systems laboratory consider smaller and simpler systems, generally referred to as instructional operating systems. This work continues in the same direction and is threefold.

First, it considers the hardware platform that is simpler and popular. Second, it argues that a minimal operating system is a viable option for delivering laboratories. Third, it presents a laboratory teaching platform, whereby students build a minimal operating system for embedded systems. The proposed platform is called MiniOS. An important aspect of MiniOS is that it is sufficiently supported with additional technical and pedagogic material. Finally, the effectiveness of the proposed approach to teach operating systems laboratories is illustrated through the experience of using it to deliver laboratory projects in the Operating Systems course at the University of Northern British Columbia.

Finally, from so little sleeping and so much reading, his brain dried up and he went completely out of his mind.

Miguel de Cervantes Saavedra, Don Quixote.

A mi madre y a la memoria de mi padre.

Acknowledgements

Foremost I would like to express my gratitude to my supervisor and friend Dr. Alex Aravind for his trust and selfless support, even he when did not have to.

I would also like to thank Dr. Alex Aravind, Dr. Jernej Polajnar, and Dr. David Casperson, who (likely unknowingly) have served as inspiration during the duration of my studies. To Alan Kranz for fabricating hardware components for our laboratories.

I thank my fellow lab mates Bolo, Kurtis with K, Dhruv, Shanthini, Rahim, and Aarthy for the good times we had in the past few years inside and outside of the lab. Also to Conan for those Tim Horton's discussions about nothing in particular. To la miss for so many coffee invitations.

Last but not least, my thanks go to Ms. Mahi Aravind for all the dinners and concern. It never went unnoticed.

Contents

List of Figures

List of Tables

Publications from this Thesis

1. 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).* ACM, New York, NY, USA, 430-435.

Chapter 1

Introduction

Operating Systems is a central topic in undergraduate computer science curricula. Comprehension of subsequent computer science courses relies on the proper understanding of the operating systems (OS) course. Whilst this is similar to many other undergraduate courses, what makes the OS course peculiar is the difficulty of delivering its laboratory assignments. Due to its complexity and scope, OS courses are delivered in several styles.

Several universities across the world deliver purely theoretical OS courses, and this is one extreme. Many universities, particularly top western Universities, offer OS courses with a heavy project component. This is the other extreme, and obviously the effective way of teaching OS because it gives the opportunity for the students to have a hands-on experience. The rest offer the OS course in between these two extremes. In the words of M. Ben-Ari:

> "Programming is the fundamental activity of computing. As such it must be a major component of courses for students of computing. Courses should not be purely descriptive; instead, they must require students to construct implementations." [4]

Nonetheless, delivering labs where students write or modify an operating system

in a semester is a challenge. Operating systems are typically large, intricate, concurrent, low-level pieces of software. Writing one requires dealing with: i) asynchronous interrupts; ii) direct access to memory and registers; iii) the inner details of the target computer architecture; iv) the size of the OS itself; and v) the concepts and ideas behind each of the different OS components. Thus, offering the same kind of practical exposure present in some other computer science undergraduate courses is, at best, impractical.

Several approaches for teaching OS laboratories have been proposed in literature. Given that concurrency and low-level programming (i.e. (i) and (ii) above) are inherent to the hardware platform programming model, efforts in the computer science education community have focused on creating smaller and simpler instructional OS (i.e. they have focused on (iv), (v), and less on (iii)). Continuing in this direction, this work takes the small-size philosophy of instructional OS further, and proposes a minimal system to deliver laboratory assignments. In addition, it attempts to lessen the difficulties that originate from programming a complex machine (i.e. (iii)). Specifically, it does so without opting for either simulated or emulated hardware, nor hiding it behind software abstraction. It instead proposes the use of less complex hardware. Then it combines everything together in a simpler platform called MiniOS, a laboratory teaching platform. Lastly, we discuss the effectiveness of the proposed approach and our experience of using it for the past few years.

Thus far the platform is comprised of the system, a guide to its design and construction, suggested laboratory assignments, and additional didactic material. Further, with the purpose of student engagement, wireless capabilities have been added to the system. Also, the work on quick integration of drivers has gone into the system as well as the didactic material.

1.1 Thesis Contributions

- Presentation of a novel approach to teach operating system laboratories
- Implementation of the teaching platform
- Development of supporting materials, expected to be released as a book.
- Testing of the proposed approach in the classrooms

1.2 Thesis Organization

Chapter 2 briefly reviews the literature related to the work presented in this thesis. It categorizes different approaches and presents where our work stands in relation to the categorization. Chapter 3 traces the origins of the main difficulties in teaching OS labs in terms of software system, hardware platform, and lack of expertise of students in OS development. Then it uses them as the basis to propose a new solution called MiniOS based on minimal software, minimal hardware, and a guide specifying the construction of the system. Chapter 4 describes the MiniOS and its architecture, discusses the embedded target platform, and elaborates on the guide. Subsequently, a set of laboratory assignments together with recommendations of its delivery are provided. Chapter 5 discusses the evaluation of the final product, and the experience in using it to deliver the laboratory projects. In the end, in Chapter 6, concluding remarks of the work and future research directions are given.

Chapter 2

Related Work

The teaching of OS lab projects can be broadly classified into four approaches: (i) those where the OS is partially or entirely simulated; (ii) those modifying or extending a full-fledged operating system, either desktop, mobile, or embedded; (iii) those where a toy operating system is built from bare metal; and (iv) those modifying or extending an instructional OS (whether they execute on simulated, emulated, or actual hardware).

Simulation based approaches are attractive as they capture high-level functionality, which can be presented in a visual and intuitive manner. Yet simulations are unrealistic, thereby limiting the learning experience. Conversely, modifying or extending a full-fledged operating system, such as GNU/Linux, does provide the experience of working with a *real system*. This is, however, at the cost of a steep learning curve, which results in students having time to modify a limited number of components in a superficial manner. It is our opinion that these two methods are inadequate, and we will not consider them further. Our views on production operating systems as teaching tools comply with those found in literature [1]. Since our approach relates more with (iii) and (iv), they are described in more detail in the remainder of this chapter.

2.1 Building a Toy OS from the ground up

Building a toy OS from the ground up involves students designing their own simple OS. Out of convenience, a virtual machine (e.g., bochs) is typically used as development platform; though it is possible, with some assistance, to have students execute their OS in actual hardware. Examples of instructional operating systems following this philosophy are the uMPS/Kaya platform[14], the TempOS platform[28], GeekOS[19], VIREOS[10], Black's OS [5], and Chadwick's OS[6]. The building of a toy OS approach has the following advantages:

- There is no pre-existing OS to assimilate;
- Building the system from the ground up demonstrates how the system fits together, thereby gaining a holistic view of it;
- Building a own OS gives a gratifying feeling (this we observed from our experience with the students and also find similar views are reported in [14]).

The disadvantage, on the other hand, is that students need to work directly with hardware that is intricate and has a steep learning curve. Complex hardware together with the difficulties of writing an OS from the ground up, leaves no opportunity to cover more than a few topics in their most rudimentary forms. Consider the case of Chadwick's OS, where four out of eleven lessons are dedicated solely to controlling the screen. The final OS then not only has little resemblance with a production OS, but is also a tiny toy—in the sense of not being developed enough to have any practical purpose. Moreover, the lack of device drivers availability is a problem for both students and instructors. Device drivers constitute a bulk of operating systems code.

Writing drivers is a difficult technical task, beyond the skill set of anyone who is not an experienced kernel developer. Without having proper drivers support, it is impossible to go ahead with projects, be they lab assignments or final projects. For instance, in [5], a lab project on user and kernel mode separation was almost impossible due to the use of BIOS for accessing I/O. Due to such complexity, some systems such as KayaOS and VIREOS have opted for running on simpler simulated hardware. Although it does bring the complexity down, it is at the cost of realism.

2.2 Modifying/Extending an Instructional OS

In this approach, students are given the task of manipulating an instructional OS; namely, adding functionality or modifying the existing one. Unlike production operating systems, pedagogic ones are more compact. That is, the number of concepts, amount of code, and technical details that must be comprehended involved in the latter case are fewer. Examples in this category are: Nachos[8], Pintos[27], PortOS[3], BabyOS[24], OS/161[18], Topsy[12], among others. There are some advantages to this approach:

- Their smaller size makes them more approachable than production operating systems.
- Interaction with hardware is not direct, as interaction occurs with pre-written lower abstraction layers.
- They (some more, some less) resemble operating systems as built in reality.

On the down side, resembling real operating systems entails complexity. Thus,

these systems deal with the issue of how much realism must be traded for simplicity. Consider, for example, the case of modifying a FAT32 file system. Students ought to have some understanding of the file format itself, the actual—often non-trivial—code used to implement it and its interaction with other system modules. On the other hand, a simpler ad-hoc file format, which lends itself to easier comprehension, is not a file format used in deployed systems. In other words, there is no single system that fits both simplicity and realism. Instructors must, therefore, select one that adequate to their teaching objectives.

Like build-your-own approaches, many instructional systems cannot be used for any purpose other than instruction; as often they do not run on actual hardware, or they simply are not developed enough. Those that can are complex systems. A survey of instructional OSes can be found in [1].

2.3 The Xinu approach

An intermediate approach that fits in the two previous categories is the one behind the idea of Xinu[9]. Xinu is an instructional OS and is peculiar in that a guide (in the form of a book) to its design and implementation is available. While its design and inner workings are detailed, its implementation is also given a rationale and demonstrated in code. This makes Xinu more self-contained, meaning, that the necessary knowledge to put the system together is part of the guide. Altogether, the book removes the mystery surrounding the OS.

Instructors may have options for students to either extend/modify the system or build it in its entirety. So, the advantages and disadvantages are a mixture of the two

previous approaches. Importantly, the system developed is not a toy, but a complete and functional operating system, and for the same reason, it is complex. In fact, it is intended to be used for advanced courses with a focus on production operating systems. Consider one of the highlight remarks from the back cover of the Xinu Book (Lynksys version):

> "Designed for advanced undergraduate or graduate courses, the book prepares students for the increased demand for operating system expertise in industry."[9]

Similarly, from embedded Xinu's website:

> "A student built operating system puts the student in the trenches of operating system development. The student will become intimately involved with the inner workings of an operating system."[32]

Hence, this approach is less suited for any introductory operating systems course. Further, the guide does not touch on any hardware-related details, leaving to students and/or instructors the task of filling the considerable gap between the OS guide and the hardware documentation. One might point out that other operating systems such as Minix[30], Kaya, TempOS and Topsy also come with a document describing the system. Yet, none is as self-contained, nor offers the amount of detail as that of Xinu.

2.4 Summary

This chapter categorized and discussed the different approaches to teaching of operating systems labs from the literature. It discussed their basis, advantages, disadvantages, and identified those approaches related to our solution: namely, modification of instructional systems, building of systems from the ground-up, and Xinu's approach as an intermediate solution. With this brief review of the related work, we next present the rationale for our approach.

Chapter 3

Rationale for MiniOS

A student-built production-like operating system that can run on real hardware is the ideal realization of the philosophy of teaching with the objective of hands-on experiential learning. Similar views are expressed in [14] and [18]. Whether the system of choice is mobile, desktop, embedded, or some other will depend on the specific instructional objectives of courses. In either case, a dichotomy exists between the ideal and fitting the workload into one semester. In this chapter, we attempt to explain this dichotomy and then provide the basis of our solution.

3.1 The issues of building a complex system

There is no need to build a labyrinth when the entire universe is one.

Jorge Luis Borges

Operating systems are complex. Mosley et al [26] point out that complexity has a direct impact on one's attempts to understand a system. They also identify the three main sources of complexity in software: *state*, *flow of control*, and *code volume*. Given the sizes of modern operating systems, as well as the topic in question (OS instruction), we focus our attention on code volume.

In general, larger systems are harder to understand. How much harder? As expressed by Dijkstra, it is still unclear:

> "It has been suggested that there is some kind of law of nature telling us that the amount of intellectual effort needed grows with the square of program length. But, thank goodness, no one has been able to prove this law. [...] As a result I tend to the assumption—up till now not disproved by experience—that by suitable application of our powers of abstraction, the intellectual effort needed to conceive or to understand a program need not grow more than proportional to program length." [11]

Modern operating system sizes are typically in the order of millions lines of code (LoC), and it is not surprising to find them in the list of the largest softwares [33]. Consider, for instance, the latest versions of the mobile operating systems Android and Symbian, or the latest versions of the desktop systems GNU/Linux, Mac OS, and Windows; all of them are composed of millions of LoC.

For this reason, the computer science education community has favoured the use of smaller instructional operating systems. That is, simpler systems in which the main purpose is to serve as a teaching tool. To put this into perspective, consider the size, in LoC, of some popular instructional systems: Minix and Xinu, with tens of thousands; Kaya OS with over 7,000; Pintos with over 5,000; and Nachos with approximately 2,500. Their smaller size enables labs to be carried out in one semester's time (some with more difficulty than others).

Instructional systems may or may not have what we consider two important characteristics: namely, being complete and functional. Complete in the sense of implementing the typical components of an OS, and functional in the sense of supporting

execution of real applications on real hardware (e.g., a teller machine system, a basic laptop, or a robotic system). Unfortunately, a complete and functional system is more realistic, and thereby, more complex. For instance, Minix and Xinu are complete and functional systems; hence, unsuited for undergraduate instruction. From those remaining, none of them are functional, and their degree of completeness varies—Nachos being the smallest yet still arguably complete amongst them due to their philosophy of minimal implementations. Incidentally, we use the term *minimal* to describe Nacho's implementation philosophy: "Our approach was to build the simplest implementation we could think of for each sub-system" [8].

Accordingly, we argue that, by means of minimal implementations, we can build a system with further reduced code volume. Moreover, we can use this reduction in size and complexity as an opportunity to:

a) Cover (i.e. implement) components that are otherwise "out of scope", and

b) Build a system capable of serving a purpose using actual hardware.

In other words, we make the case for a minimal—yet complete and functional—instructional operating system. Thus far we have discussed the difficulties of dealing with OS software. Now, we consider another source of complexity—the target hardware platform.

3.2 The issues of complex hardware

"In order to be creative one must first gain control of the medium. One cannot even begin to think about organizing a great photograph without having

the skills to make it happen." Gerald J. Susman

Present time computers are intricate pieces of hardware. Manuals detailing the functionings (from a programmer perspective) of a modern 32- or 64-bit processor add up to at least a few thousand pages. To that, one must add the documentation detailing the functioning of the rest of the computer hardware, e.g., interrupt controller, BIOS/UEFI, timer, real-time clock, and others. For these reasons, writing non-trivial bare-metal applications (such as operating systems) is a technical, tedious, error-prone, and laborious task. One ought to know the precise inner workings of the computer if she hopes to direct it to do anything. Even though OS courses are customarily preceded by architecture or organization courses, these inner workings are often too advanced, and there are too many details to be covered in their entirety. Additionally, the machine exposes a programming model of asynchronous interrupts. Concurrent code accessing arbitrary memory and registers is one of the most challenging code students will encounter during their studies.

Then, how can a student possibly aspire to build an operating system, even a simple one, in one semester? The answer is simple—they cannot. For this particular source of complexity, solutions have been proposed in literature. One solution is to build the system for a hardware simulator or emulator that is simpler to interact with. This is advocated and demonstrated in Kaya OS, OS/161, VIREOS, PortOS, and Nachos. For example:

> "Simulators 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."[14]

A second solution is to abstract away hardware via a software layer. While this is indirectly followed by any instructional OS not meant to be built from bare metal, GeekOS explicitly follows this approach:

> "Working at the hardware level has two main disadvantages. First, hardware devices can be tricky to program correctly. A more fundamental problem is that debugging kernel code running on real hardware is difficult, even for experts. The contribution of our work is to show that both of these difficulties can be overcome without requiring heroic measures from students or instructors. We have implemented a tiny OS kernel, called GeekOS, which provides a sufficient abstraction layer over the hardware to hide the genuinely difficult details."[19]

A third solution is to compromise on the level of sophistication of the system, so as to simplify the technical (hardware) details required to build it. This is put into practice in Black's OS and BabyOS, where students build a toy OS from bare metal.

We are of the opinion that exposing the students to real hardware is not only essential for a holistic understanding of the system, but also increases their engagement. Similar views are expressed by Pfaff et al[27]. Therefore, we consider only the latter approach to compromising on the level of sophistication. Unfortunately, such compromise results in a system that does execute in real hardware, but it is far from being complete and/or functional.

Yet, we argue that by targeting a simpler real hardware platform, we can decrease the technical knowledge required to build an instructional system; then, use that as an opportunity to build a complete and functional system. In other words, we make

the case for a minimal—yet complete and functional—instructional operating system for a minimal hardware platform.

Instructional OSes achieve simplicity by trading the capabilities of full-fledged real systems. Next, we elaborate on it.

3.3 A minimal instructional OS for a minimal platform

If realism must be traded for simpler minimal implementations, the question that follows is, where is the ideal trade-off point between one and another? This is a difficult question, and it is (directly or indirectly) explored in each and all of the different instructional operating system proposals. For instance, Holland et al elaborate:

> "For teaching, a certain amount of realism is desirable. Too much realism, however, becomes both too complicated and, sometimes, realistically painful. [...] [R]eal OSes are immensely large and complicated, and are full of complexities and constructs for coping with real-world issues that have little instructional value."[18]

Liu et al also elaborate:

> "In the process of using BabyOS, we found that it is really difficult to make tradeoff between realism and simplicity. A certain amount of realism is desirable, otherwise BabyOS feels like an unreal OS. Too much realism,

however, becomes too complicated and, student would fail to finish their projects."[24]

Even though we do not know where the ideal trade-off point resides, it is our intention to explore it by implementing a minimal instructional OS for a minimal hardware platform, which we call MiniOS.

Real being impractical, we focus on the minutiæ that can preserve "relevant realism" in trade of "less relevant realism" (as far as undergraduate instruction goes). In particular, MiniOS is complete and functional. It is targeted for a real hardware platform; and it follows the design, layout, and mechanisms of real systems. Meanwhile, fault tolerance, robustness, efficiency, reliability, sophistication, and other attributes in deployed systems are not considered.

To put it bluntly, whilst MiniOS should not be deployed as part of an aircraft computer or an X-ray device, it is perfectly suitable for less important applications, such as a gardening system, or an unsophisticated robot—and such system, we believe, is well suited for instruction.

Thus far we have used the term *real hardware* generically, now it is time to specify a target platform.

3.3.1 A minimal embedded hardware platform

We use the term minimal hardware to refer to those computers with the least amount of sophistication still capable of hosting an OS. For the sake of exploration, we have selected what we consider to be one of the smallest among them; more specifically, a

32-bit ARM low-end embedded platform. This choice is partly arbitrary and partly influenced by ARM's popularity in the mobile and embedded systems industries.

It is worth noting that the term *embedded* does not imply simplicity. While there exists basic 8-bit microcontroller (MCU)-based embedded computers (e.g. a coffee maker's computer), there too exists sophisticated 64-bit microprocessor (MPU)-based embedded computers (e.g. an industrial robot's computer).

Despite the fact that some instructional systems, such as Minix, Xinu, and BabyOS are targeted for (or have been ported to) embedded platforms, they differ from our philosophy of minimal hardware. In fact, to our knowledge, there is not an existing instructional OS with similar views on hardware.

It is also important to clarify that MiniOS is not intended to be an embedded production OS. Like desktop systems, embedded production OSes are complex. They tend to be plagued with intricacies that make them adequate for deployment in life-critical applications such as aircraft and military. A representative sample, and in the smaller side of the size espectrum, is FreeRTOS [25], which, intended for low-end embedded platforms, has over 9,000 LoC [29].

A low-end embedded OS may seem as an over simplification, and naturally one raises the question of whether such a simple system has any instructional value outside the embedded systems realm.

3.3.2 A low-end embedded OS as a teaching tool

The purpose of MiniOS is not to serve as a tool for teaching embedded systems, but to serve as a tool for teaching general principles that apply to operating systems. In fact,

MiniOS is not well suited for teaching labs in embedded systems, as embedded-specific details are deliberately overlooked. With few exceptions where it is impossible, it is emphasized how they contrast with general purpose computers. Consider, for instance, the case of a MCU-based low-end embedded platform (a Von Neumann architecture) with Flash as program memory; it must be brought to the students' attention that general purpose systems do not, customarily, have non-volatile program memory in their address space. Thus a boot-loader for a MCU will be different than one for, say, a desktop computer.

Fortunately, the similarities are greater than the differences, and this is why we believe a simpler low-end embedded system can be used as a teaching tool. That is, for a course with no intention of preparing students for real-world OS development (whether embedded, desktop, or other).

One important benefit of working with MCU-based embedded platforms is the availability of device drivers. Hardware manufacturers typically release open source bare metal middle-ware (mostly drivers) to be used on their platforms.

Finally, we argue that, recently, there has been a switch from traditional desktop systems to mobile and embedded systems (e.g. internet of things and wireless sensor networks). An embedded instructional system with wireless capabilities can be a tool for introducing students to the latter. A similar argument is expressed by Atkin and Sirer[3].

An equally important aspect of MiniOS is its guide. It covers building the system from nothing, and it is described in the following section.

3.4 From the ground-up: a guide to MiniOS design

"The devil is in the details."

Popular Saying

MiniOS is intended to be built from the ground up, on bare metal. For this, a guide to its design is primary. In a comprehensive and thorough manner the guide must—step by step—detail the construction of the system from nothing. All the technical details dealing with the hardware, the compiler, as well as OS concepts and their specific implementations should be covered, including details such as exceptions, memory mapped IO, linking of relocatable code, calling conventions, memory segmentation, and context switching.

Other instructional systems also advocate for the use of a guide or manual [19, 12, 14, 28, 10, 30, 27, 8]; some with more details and code than others. None, however, go to the amount of detail (instruction) that we consider necessary for building an OS from the ground up. (XINU is the exception; the amount of instruction offered as written material in [9] is near to what we advocate for.)

Guzdials [16] argues that the amount of instruction matters when teaching computer science to beginners. In particular, "putting introductory students in the position of discovery information for themselves is a bad idea." Although this argument is given in the context of introductory programming (100 level courses), the instruction in question (operating systems and computer architecture/organization) is introductory, as both systems programming and programming at such low-level of abstraction are substantially different from what students have encountered in preceding courses.

From experience we have noticed that, at this introductory stage, most students lack the experience, the patience, and the right approach to meticulously construct and debug low-level systems' code. Moreover, they are faced with programming patterns and tricks specific to the machine's programming model. While many of these patterns are simple and of common use, it can be difficult to re-invent them if one has never encountered them before; in contrast with higher-level programming, bugs manifest differently (typically the CPU faults and does nothing) in low level. Code is highly dependent on a great number of machine-specific details, all of which must be set correctly, and access to raw memory requires precise knowledge of its organization and how instructions access it. Moreover, it is practically impossible for students to obtain all of the required details for OS construction from the thousands of pages included in the documentation, for they are not at the level of understanding the technicalities. The end result is that students are prone to get hopelessly stuck.

Consequently, we consider that a guide demonstrating how to build the system from the ground-up: as well as specifying, in a comprehensive manner, the technical details relevant to OS writing is a necessity for the delivery of OS labs.

3.5 Summary

This section elaborated on the origins of the difficulties behind teaching operating systems labs. It explained the complexity that students must undergo when dealing with the software system, the hardware platform, and a branch of computer science for which they lack skills. From this we derived the foundation of our solution, which is based on minimality principles. With this background, we now proceed to describe our proposed system called MiniOS.

Chapter 4

MiniOS—Proposed OS instructional platform

The proposed OS instructional platform consists of the system, the target hardware, and its construction guide. This chapter describes them and gives a set of suggested laboratory projects, as well as recommendations for their delivery.

4.1 The system

First we present the high level architecture of the system, and then describe the different parts that constitute the system.

4.1.1 Architecture

From an architectural point of view it is unclear the parts that must be included in a presumably minimal, complete, and functional operating system. It cannot be composed of too many parts (layers or modules) as to become complex, nor it should have too few as to be incomplete or non-functional. Our approach on this is to incorporate components typically found in production systems, and offer the choice

of what components make it into the system. Specifically, the system is built as a set of loosely coupled modules categorized in base modules and optional modules, as shown in Figure 4.1.

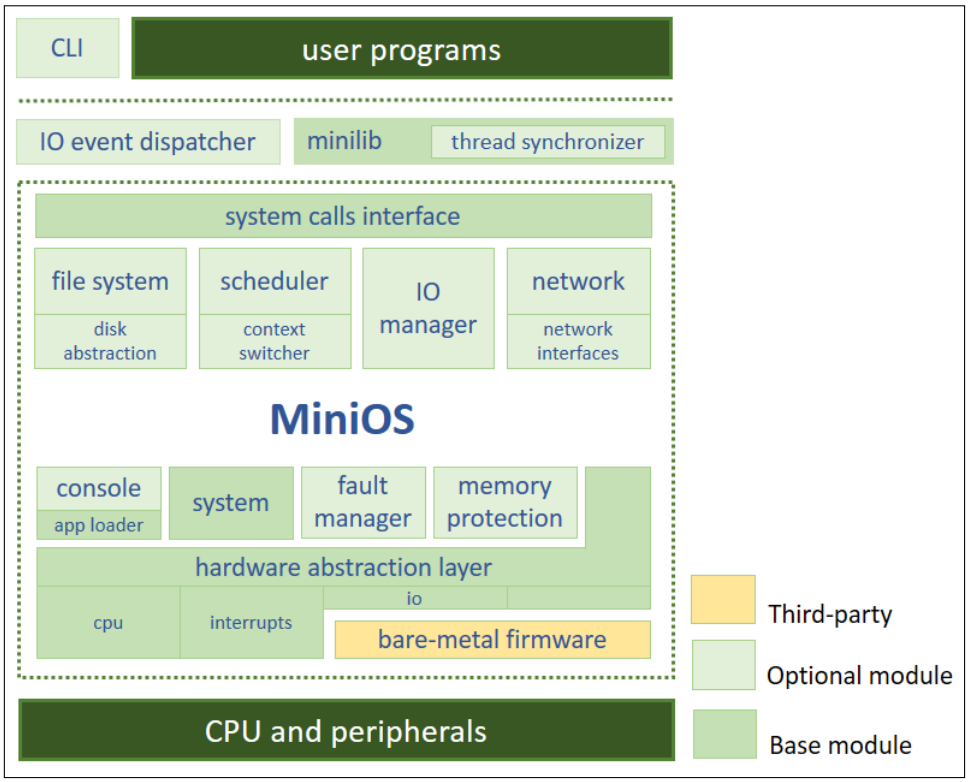


Figure 4.1: Architecture

As their names suggest, *base modules* form the foundation of the system and must be implemented, whereas *optional modules* add specific functionality that may or may not be integrated in the system. This configuration gives lab instructors and students the flexibility to start with a minimal base and add modules to accommodate to their instructional objectives. Complying with the minimality principle, the system has as few lines of code and as few components as possible.

From a design perspective, we classify the modules into two types: *primary* and *secondary*. *Primary modules* represent an identifiable OS component: hardware ab-

straction layer (HAL), fault manager, memory protection, file system, scheduler, IO manager, network stack, system calls interface, IO event dispatcher, minilib, thread synchronizer, and command-line interface (CLI). *Secondary components* offer some abstraction or functionality but do not represent an OS components: context switcher, disk abstraction, network interfaces, app loader, IO, CPU, and interrupts. Every component is mapped to a source file of the same name. There are as many C or assembly files as there are components in the architecture.

From a software engineering perspective, a modular architecture has additional benefits. First, it improves modifiability of the system. It allows students to add or remove modules with little or no modification of others. Second, it improves local reasoning, hence aiding our main objective of making the system easier to comprehend. Such design is typically achieved with support from a programming language. However, since the system is written in C and assembly, we rely merely on discipline. Particularly, we strongly advise students to keep state confined to the scope of a module, and let module interaction occur only via interfaces; practices, which we demonstrate throughout the construction guide. It is worth noting that, although instructional systems are more or less designed in this manner, often modules end up keeping global state used by other modules. More importantly, we want to make it explicit that these software engineering practices are essential for our purpose.

An important aspect of MiniOS architecture is that, unlike production systems, device drivers are in direct contact with hardware. This means, the system is built on top of them, instead of them being part of the system itself. Although some instructional systems follow this design for simplicity purposes, we do it explicitly to support integration of open source third-party firmware that is often only available as bare-metal. With this small design choice, MiniOS benefits from available code that

are from chip vendors and/or embedded systems enthusiasts. With this higher level description, next we will describe the individual components.

4.1.2 Components description

We start with the base modules.

- *HAL:* This is the lowest layer of the system and it is responsible for providing sensible machine-independent abstractions to upper layers. Particularly, it implements three abstractions: CPU, interrupts, and IO.
- *System:* System is central to the rest of the modules, and is in control of all the system-related tasks, such as system initialization and kernel panics. Additionally, it offers implementations of various data structures to aid in the development of the kernel.
- *Application loader:* This module is responsible for the loading of applications from the SD Card. It is used for either automatic loading of pre-defined applications after OS initialization, or in the presence of the CLI, for executing applications by name.
- *System call interface:* After configuring the CPU to run in user mode, the system calls interface serves as the only gateway to the system. Invocation of system calls is via software interrupts.
- *Minilib:* This small library module sits in between applications and system calls. Minilib's purpose is to:
  - Wrap up system calls and presents user applications with a more intelligible interface.

  - Provide support for buffered IO operations in the presence of the IO manager.

Now we describe optional modules.

- *Fault manager:* It is a small module whose only task is to raise kernel panics on the occurrence of CPU faults (e.g. div by zero fault).
- *Memory protection module:* This module protects kernel code and data from code running in user mode. It restricts applications from accessing specific parts of memory, generating a segmentation fault if boundaries are violated.
- *Thread synchronizer:* Albeit part of minilib, thread synchronizer is a module on its own. It contains implementations of thread synchronization mechanisms: lock, semaphore, monitor, and barrier synchronizations.
- *Scheduler:* The scheduler is a limited, but functional, priority-based pre-emptive thread scheduler. It supports a fixed number of threads with fixed stack sizes. While termination for a given thread is supported, freeing of its memory is not (mainly to avoid handling complex memory details). Threads can yield, can signal other threads, and can sleep. For portability, platform-dependent code for context switch is part a context-switcher, and not the scheduler itself.
- *File system:* A functional operating system must have a file system to start with. MiniOS uses a part of FatFS [7] as file system. FatFS is a small FAT file system for resource-constrained devices.
- *CLI:* The command-line interface is a shell whereby applications can access a small number of kernel services. Some commands are, for example, *ls*, *cd*, *cat*, and *netstat*; minimal versions of GNU/Linux's commands with the same name.

- *Network:* As for networking capabilities, the network stack supports a very simple, inefficient, but functional network protocol over IEEE 802.15.4. Namely, it uses a flooding algorithm to form a network of ad-hoc connected devices. To avoid dependencies, the network stack purposefully overpasses the IO manager and handles its own buffers and radio interrupts.

- *IO manager:* The IO manager controls access of I/O devices. When interrupt-based devices notify the system of available data, it is responsible for:

  - Placing the incoming data in an intermediate buffer accessible to both minilib and the IO event dispatcher.

  - Notifying the scheduler of new incoming IO data.

- *IO event dispatcher:* The IO event dispatcher enhances the system with *IO events*. Whenever the scheduler is notified of new incoming IO data, the event dispatcher runs and executes the corresponding user-level event handler. Unlike other modules that can be implemented on top of base modules, the IO event dispatcher requires the scheduler and the IO manager to be part of the system. For a more concrete idea, consider the sample program in Listing 4.1.

Listing 4.1: Sample MiniOS application

```
#include "minilib/thread.h"
#include "minilib/oled.h"
#include "minilib/network.h"
#include "minilib/sensors.h"
#include "minilib/led.h"
#include "minilib/ioevents.h"

void salute_thread( void* params ){

    thread_set_priority( (uint32_t)params );

    while( true ){
        //print salute to USB
        usb_write( "Hola, soy %s       \n", thread_get_current() );
        thread_sleep( 200 );
    }
}

//Layer-2 frame received event handler
IOEvent net_frame_received( NetFrame* frame ){

    //echo
    net_mac_send( frame );
}

int main(){
    //Create salute threads
    thread_create( salute_thread, "Mariana", 128, THREAD_PRTY_MIN );
    thread_create( salute_thread, "Cafe", 128, THREAD_PRTY_MIN );

    uint32_t state = 1;

    while( true ){
        //print sensor information to OLED
        oled_write( "Light level (%%): %d \n",
            light_read(LightScale1to100) );
        oled_write( "Temperature (C): %d \n", temp_read() );

        //blink LED0
        led_set( Led0, state++ % 2 == 0 ? LedOn : LedOff );

        thread_sleep( 500 );
    }
}
```

This program is composed of four threads, one of which is main. Two of them print their name approximately five times a second over a CDC USB connection;

one waits for an incoming network message and echoes it back to the same sender; and main prints sensor information on the OLED screen and blinks an LED approximately every half second.

An operating system works closely with a specific hardware. The following section discusses the target hardware platform.

4.2 The target hardware platform

Among all the different available ARM processor cores on the market, the Cortex-M series are those with the least sophistications that still offer support for operating systems. Among them, we have opted for the Cortex-M4, which was the most sophisticated in the Cortex-M series at the time MiniOS was initially conceived. Some of these OS-supporting features are software interrupts, memory protection, different CPU modes (kernel and user), separate user and kernel stacks. In fact, the only missing feature to fully support a conventional OS, capable of executing applications, is a memory management unit (MMU).

Cortex-M cores are only available in micro-controller units (MCUs), and because a MCU by itself is of no use, a MCU prototyping (evaluation) board must be used. Although it is possible to carry out labs with tailor-made hardware, an off-the-shelf board has its advantages. First, there are available device drivers from manufacturers. Second, these boards typically integrate an on-board chip debugger and programmer, thereby eliminating the need of an expensive JTAG emulator that does the same. Third, they can be purchased by anyone interested in taking or delivering the course. Lastly, being official boards, they integrate seamlessly with manufacturers' develop-

ment tools.

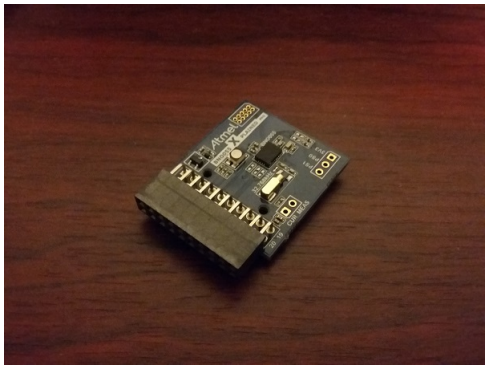
A variety of MCU prototyping boards exist in the market from different vendors. Based partly on its low cost, and partly in nothing in particular (as they all are quite similar), we have selected the *Atmel SAM4S Xplained Pro Starter Kit*. Its main board runs at 120 Mhz, and together with its three expansion boards integrate enough peripherals for laboratory projects. They include a small OLED screen, buttons, LEDs, a light sensor, a temperature sensor, a microSD card slot (and the microSD card), a USB device port, an on-board 256 MB Flash memory, and exposed pins for on-chip peripherals such as GPIO, UART, USART, ADC, PWM, I2C, and SPI. Figure 4.2 shows the main SAM4S board and its daughter boards, together with the *REB233 board* (acquired separately) for IEEE 802.15.4 connectivity. This is the hardware assumed by the construction guide.

Figure 4.2: Target Platform: SAM4S Starter Kit and REB233 radio

It is worth noting that in a previous offering of the course the IEEE 802.15.4 Xbee[20] from Digi was used as radio. However, students had problems with the extra wiring required, and a few Xbee modules were burned in the process. Being

plug and play, the REB233 board is expected to serve the same purpose without any wiring.

Optionally, the *BNO055 absolute orientation sensor* (Figure 4.3 (a)) can be used as additional hardware to enable applications related to robotics, navigation, and others where tracking of pose or motion is desired. It is a low-cost absolute orientation sensor that integrates an accelerometer, gyroscope, and magnetometer to provide raw data and a hardware-calculated orientation in euler angles. Although the final release of MiniOS does support it, it is only mentioned in the guide as complementary material; that is, it is not required for completion of laboratories.



(a) BNO055 orientation sensor

(b) Hardware for possible a GUI laboratory

Figure 4.3: Additional optional hardware

Lastly, it is possible to seamlessly add support for touch screen using the Atmel maxTouch Xplained Pro (Figure 4.3 (b)). Due to its high cost, it is not supported by MiniOS. Still, it represents a good option for a GUI laboratory as it plugs directly as an expansion board, and drivers are available.

Being an embedded platform, development of software is somewhat distinct. The following section attempts to offer more details in this regard.

4.2.1 Development Environment

Clearly one cannot (easily) use the system's target platform to develop the system itself. Instead a separate *host computer* is necessary for development of the system. In particular, using a cross-compiler, first the source code is compiled to an executable in the host. Then, the executable is flashed to the target's program memory by a flashing tool. Finally, for debugging, an on-board hardware debugger interfaces with software in the host to enable source-level debugging. All of the different host-side software tools, including the GNU toolchain are integrated in Atmel's IDE: Atmel Studio. Communication between the target platform and host tools is via USB. Figure 4.4 depicts the described programming environment. Incidentally, Atmel Studio was built with Microsoft Visual Studio Shell. So, the programming environment is the same as that from Microsoft Visual Studio.

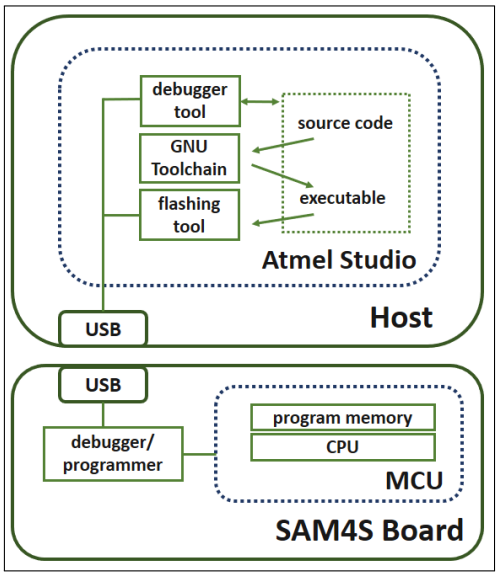


Figure 4.4: MiniOS development environment

The Atmel debugging facilities, when used correctly, allow debugging of firmware

running in the MCU as if it was a regular desktop application. It allows pausing (possibly at breakpoints) of the CPU for inspection and modification of memory, registers, IO interfaces, and source-level variables (including not primitive types). It also allows to step through both assembly and C code, as well as dis-assembled code. Figure 4.5 shows a screenshot of a sample debugging session.

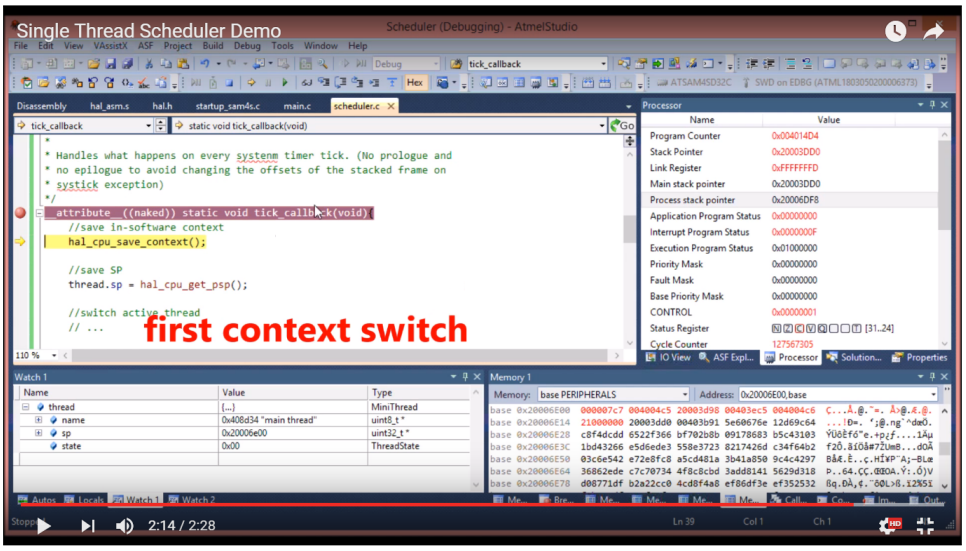


Figure 4.5: Sample debugging sessions

The final piece in the development platform is the system's guide to its construction, which is discussed in the following section.

4.3 The MiniOS Book

The idea of the MiniOS guide (or book) is to:

- Cover in-detail all the technical material that is necessary to build MiniOS.
- Guide students in the process of developing it themselves.

Consequently, the guide is intended to be self-contained, in the sense that a student could rely solely on it to build the system (characteristic not present in other similar OS books). The style of the guide was initially inspired by the tutorial *Write Yourself a Scheme in 48hrs* [31], and later by the more textook-like style of the Xinu Book [9]. Our guide is divided in two parts, as shown in Table 4.1.

Table 4.1: Book Layout

PART I (HW ARCHITECTURE)	**PART II (SW SYSTEM)**
1. Introduction	1. Basic IO and Booting
2. Instruction Set Architecture	2. Hardware Abstraction Layer
3. Memory	3. System Calls
4. IO	4. Fault Manager
5. Stack	5. Memory Protection
6. Interrupts	6. Scheduler
	7. IO events
	8. Thread Synchronization
	9. Network Stack
	10. Command-Line Interface

The intention of the first part is to instruct on computer architecture using the ARM Cortex-M4 and the SAM4S board. The second part is dedicated entirely to the system, and it assumes some working knowledge of what is covered in the first section. Ideally, a student should complete the first section of the book, and then engage in building the system. However, if this is not the case (as we have experienced), working knowledge of a different computer architecture suffices. At worst, students will take extra time to learn certain Cortex-M4 technicalities. Importantly, all these required technicalities are available for consultation in the architecture section, and when used in the systems section, they are referenced.

An important aspect of the guide is the great amount of details offered. This is because it was written with the purpose to not leave students in the situation of discovering neither advanced topics nor topics pertaining to other subjects by themselves.

Among others, it covers topics and information related to data structures, drivers, CPU, peripherals, the SAM4S board, the SAM4S MCU, the C language, assembly, the linker, and even programming patterns that are particular of systems or low-level programming. For instance, the guide explains and demonstrates the following: how to use callbacks to push data (coming from interrupts) from a lower layer to an upper layer; how to load a pre-compiled application from permanent storage to RAM for execution; how to write a linker script; how to do context switch; how to change CPU privileges; where in the documentation to find the mapping between physical pins and logical IO bits; and so forth. Some of this information is too technical or advanced to be left for discovery, and some does not pertain to OS instruction per se. Appendix 1 shows an excerpt of the Scheduler Chapter.

To put it in perspective, consider the analogy of an engineering course with the objective of teaching about principles of motor vehicles (say, their inner workings). One could have students designing, manufacturing, and putting together every single cam and piston of the motor; then going ahead in designing their own engines, manufacture them, and assemble them together; then continue to re-invent techniques and mechanisms that are otherwise well-known and of standard use in assembling of cars; and then let them teach themselves how to operate machinery that they will need. Alternatively, one could provide all the working pieces, a demonstration of all the techniques and methods they will be needing, partial solutions to the parts that are known to be difficult, and have students assemble the car. Assembling a car seems a task already difficult on its own to be adding more to it (unless the purpose of such course is to prepare students for automotive design). In other words, more than writing an OS from the ground up, we are looking for students to assemble one, from the ground up. (The analogy, of course, is not perfect, but should reflect what we are looking for in the MiniOS guide.)

Importantly, most pieces of code that are given, are not just given, but derived, meaning, the book explains the steps in obtaining it from documentation or other assumed background knowledge. This gives interested and motivated students the tools to modify those parts, should they want to (e.g. for a final project).

Additionally, a secondary device driver integration guide was developed. This smaller guide demonstrates the process of integrating third-party drivers, and shows working sample code. It can be challenging to write working code for an IO peripheral out of poor, and often buggy or incomplete, third-party documentation. Appendix 2 shows two sample entries from the driver guide.

In addition to text material we have prepared demonstrative videos. These are videos made to strengthen the text material, by showing explained concepts, techniques, processes, solutions to labs, or running sample driver code. For example, Figure 4.6 shows a debugging session right before a system call.

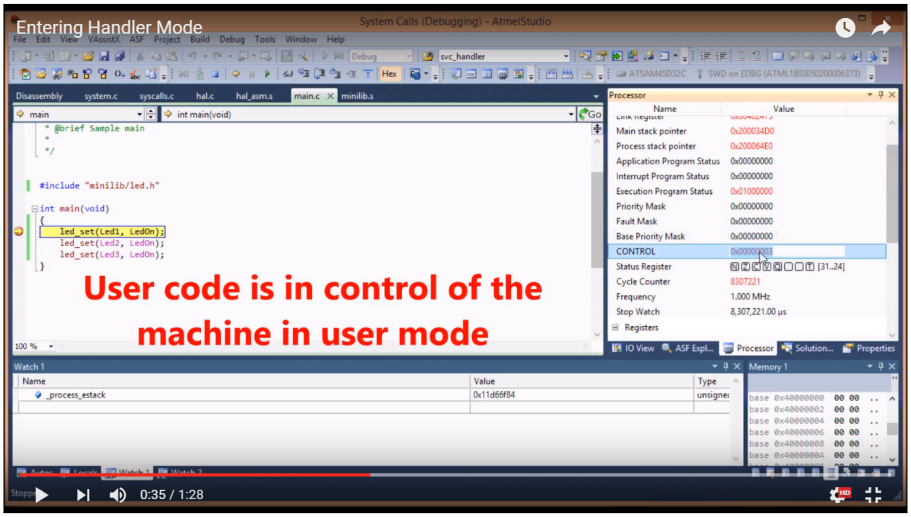


Figure 4.6: Video demo: Entering kernel mode

Within the video the control register is highlighted to demonstrate that, in fact, the CPU is in both user and unprivileged mode. Upon execution of the software

interrupt, it is shown again, but this time specifying kernel privileged mode.

Lastly, we would like to emphasize that all this extra instruction goes in accordance with what is argued by Guzdial et al in [16] in favour of strong instructional guidance for novice learners. In particular, he mentions that there is strong evidence that the minimal guidance approach we typically use in computer science instruction is inadequate. One can argue that operating systems and computer architecture are typically second and third year courses, and therefore students are not novice programmers. While this is true, students are still considered novice learners from a low-level programming and OS development perspective.

Based on the described system, hardware platform, and guide thus far, the next section presents suggestions on how to accommodate the material in actual laboratory projects.

4.4 Laboratory Projects

There is a total of twelve labs, with different suggested durations. The first lab is an optional short introduction. The next two labs are also short, and, since they involve base modules, they cannot be skipped nor their order can be altered. The remaining eight are optional, and most of them can be implemented regardless of order. In case a module is considered to be good to have, but not of interest as to dedicate a lab to it, there is the possibility to hand it in to students. For instance, the fault manager can be a useful module to have as it outputs human-readable messages when the CPU faults, and it could be given to students.

4.4.1 Lab 1 - Basic IO and Booting

In this lab students are introduced to the booting process, the use of third-party firmware as basic input-output, and the programming environment (including debugging facilities). The recommended time for solving this lab is one week and is optional, albeit recommended. Another way of looking at this lab is that it enhances bare-metal applications with bare-metal firmware, as depicted in Figure 4.7.

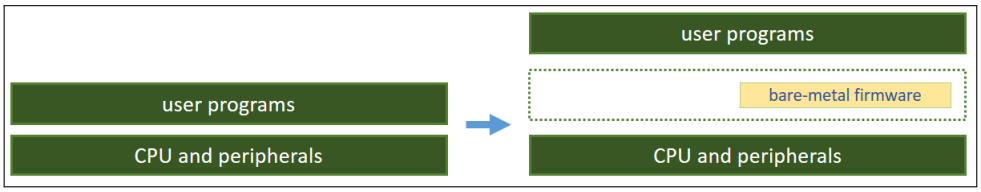


Figure 4.7: Architecture Goal for Intro Lab

The learning outcomes for this lab are to familiarize students with the development environment; to provided some guidelines on how to make efficient use of the debugging facilities; and to show the process of integrating third-party firmware to be used as basic IO.

4.4.2 Lab 2 - Hardware Abstraction Layer

For this lab students write the HAL, and the system module. Some of the implementations expected from this lab are, for example, an IO device abstraction composed of a read function and a write function, an abstraction for registering callbacks of interrupt-based IO, among others. The recommended time for solving this lab is one week, and it is mandatory. Figure 4.8 shows the result of completing this lab.

The objective of this lab is to give some insight and hands-on experience on inter-

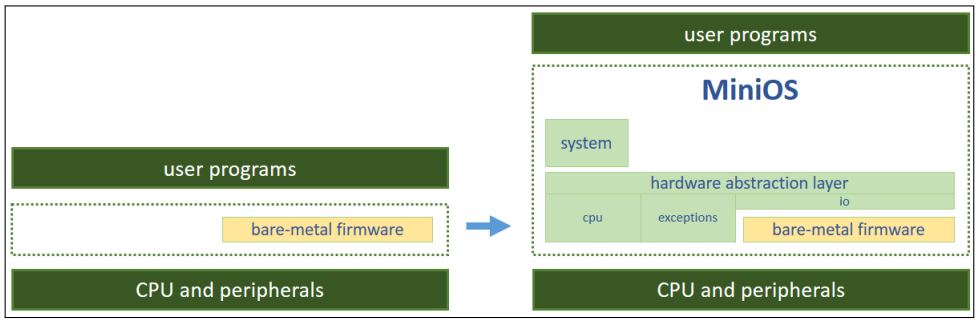


Figure 4.8: Architecture Goal for HAL Lab

action with bare-metal IO peripherals, and in the process convey students the importance that a) abstraction plays in development of the system, and b) the repercussions that a HAL has in portability.

4.4.3 Lab 3 - System Calls

Provided hardware-specific information on how to establish a kernel and user mode separation, students must add code to support software-interrupt based IO system calls via minilib. The separation is made even clearer by splitting compilation of OS and app. MiniOS is compiled and flashed to the MCU, while applications are compiled and moved to an SD Card from where they are loaded into RAM and executed. Since loader code is given, students are asked to write a rudimentary version of MiniOS CLI that supports listing of files and execution of applications only in the SD Card's root folder.

Optional tasks involve buffered output: implementation of a line-buffered oled_write function together with a a flush function. In the process, the inability of user code to directly access data from interrupt-based input is emphasized; although nothing is done about it until later labs. The recommended time for solving this lab is one or two weeks.

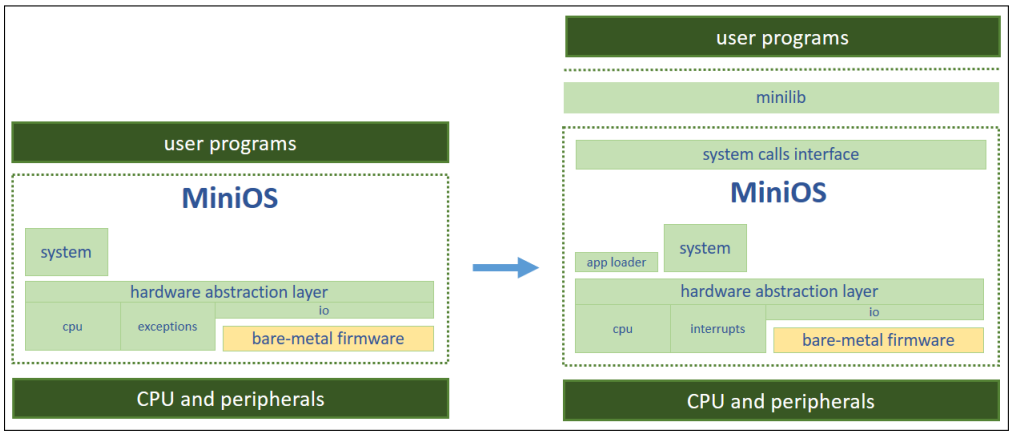


Figure 4.9: Architecture Goal for System Calls Lab

Upon completion of this lab students are expected to have some insight and working knowledge of:

- The separation of kernel from user applications and the mechanisms used by operating systems to interface them both.
- The role and place of libraries such as the GNU C Library in an operating system.
- The limitations of poll-based IO.
- Buffered IO.

4.4.4 Optional Lab – Fault Manager

The fault manager is another short lab. Here students are given guidance on CPU faults, and are asked to add support for fatal system errors—the mini black screen of death. If both memory protection and the CLI are in part of the MiniOS version for this lab, a more complex task involves termination of the offending application

and continue of execution. The recommended time for solving this lab is one week, or two with the additional task. For this lab, students are expected to gain insight as to what causes fatal system errors in computers, and have some experience in the process of handling and reporting them.

4.4.5 Optional Lab – Memory Protection

For this lab students must implement memory protection to prevent user code from accessing system code and data in memory. If the fault manager has been implemented, an extra task of enabling segmentation faults is available. The recommended time for solving this lab is one week, or two if thread protection is included. The idea of this lab is to supply students with insight and working knowledge on the use of memory protection to prevent bugs and malicious code to mess with the system, as well as to let them experience first hand what a segfault is.

4.4.6 Optional Lab – Scheduler

The scheduler is perhaps the most technically challenging lab. Starting from the system timer, a single-threaded scheduler and a yield function are derived and demonstrated. Available tasks include extending it to support multiple threads, priorities, round-robin scheduling, a sleep function, thread signalling, and different scheduling policies, among other tasks. The recommended time for solving this lab is two weeks, or three if extra tasks are added. The learning outcomes for this lab are to provide students with experience of the obscure inner workings of a thread scheduler, to let them experience first hand how sharing CPU is made possible by a set of small clever tricks done by the operating system; also, to get some working knowledge on a) im-

plementation of different scheduling policies; b) how different threads queues can be used to support sleeping threads, priorities, and thread signalling, among other tasks.

4.4.7 Optional Lab – IO events

In this lab students write the IO manager and the IO event dispatcher to add support for user-level run-to-completion IO events. Every time new data is received from interrupt-based IO, the IO manager stores it in intermediate kernel buffers and notifies the scheduler to wake up and run the event dispatcher threads. This is a short lab, and its recommended time is one week. On completion of this lab, students are expected to have an understanding of a) the implementation of events from threads; b) the benefits of a hybrid thread-event approach.

4.4.8 Optional Lab – Thread synchronization

Here students implement thread synchronization mechanisms: locks, semaphores, monitors, and barrier synchronizations. The recommended time for solving this lab is one week or two depending on the number and complexity of the mechanisms to be implemented. At the end of this lab students are expected to understand, from an implementation perspective, synchronization mechanisms used in multi-threaded programming.

4.4.9 Optional Lab – Network Stack

A MAC layer interface is delivered as part of this lab's material. Students must, then, use the trickle algorithm [21] to enhance the system with network capabilities. A more complicated task includes writing a host application that transmits an executable, having a node receive it and execute it. Since trickle is straightforward to implement, the recommended time for solving this lab is one week, or two with more complicated tasks. The objective of this lab is to demonstrate, in a rudimentary manner, how computer networks are built out of layer-2 point-to-point communication; also, to show the difficulties of a) providing applications with networking services, and b) dealing with unreliable wireless communications.

4.4.10 Optional Lab – Console and CLI

In this lab students write a either a console or a CLI (or both). The console is launched on system start up and enables:

- To print information during system initialization;
- User login;
- Execution of basic commands; and
- To browse and execute applications stored in the SD Card.

The console is internal to MiniOS. System initialization messages include CPU speed and peripherals found. More advanced features involve basic managing of user accounts, and enabling applications to exit and give control back to the console. The

CLI is an application that runs in user mode and allows similar functionality. More advanced tasks include the implementation of privileges for user accounts; the writing of a host terminal that gives it a more traditional feeling; or a host panel board that shows sensor information. This is a short lab and the recommended time for solving this lab is one week. The purpose of this lab is to demonstrate how the command line interpreter fits in with the rest of the system.

4.4.11 Final Project

As final projects, students may form teams and create something of their own. Any idea involving an embedded OS, or extension of the OS itself are eligible choices. Unlike all the remaining labs, this project has no rigid specification. It is open ended and students are encouraged to implement something of their interest. A complete version of the system can be handed in to those teams who need it. In the end, exact specifications differ depending on instructors' preference. The idea is for students to put all acquired knowledge to practice and hopefully deepen their knowledge in some specific OS aspect of their choice. This concludes the presentation of the instructional platform.

4.5 Summary

This chapter provided implementation details of all the parts constituting the instructional platform: in particular, the modular system's architecture, the MCU target platform, and the accompanying book. Finally, it presented labs with specific teaching objectives, suggested completion time, and suggested assignment tasks.

Chapter 5

Evaluation

The idea of MiniOS was not to replace other instructional systems, but to create an alternative system that could adhere itself to the already existing set, more precisely, it was meant to be a small, complete, and functional MCU-based system that could be used for teaching operating systems concepts, while lessening students' struggle.

Due to our policy of minimal implementations and minimal hardware, we were able to write the entire system in only of less than 500 lines of C code and less than 250 lines of assembly. Following the principle of not placing students in the position of discovering new non-relevant information, we have developed a self-contained book covering the construction of the system. All this together has enabled MiniOS to be:

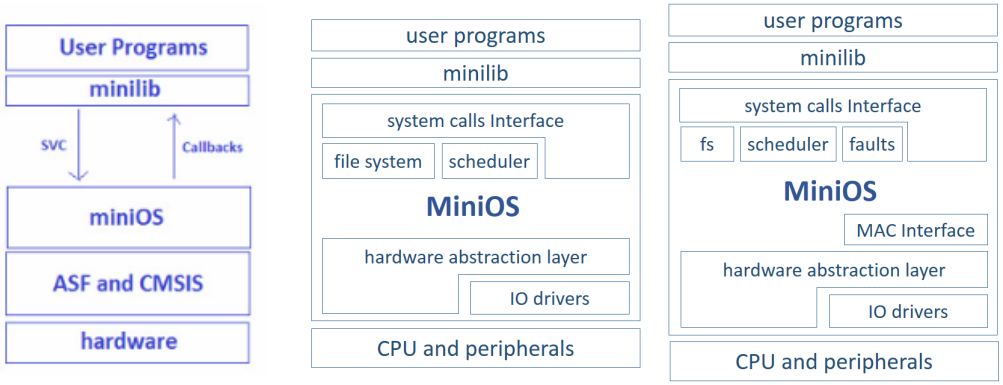
- Functional and complete.
- Small.
- Built-from the ground up.
- Simple enough to reduce students' struggle in building an instructional system.

The last objective of serving as a tool for teaching operating systems concepts is left unanswered. Instead we report on our experience in using MiniOS to teach the laboratory component of a course in operating systems.

This chapter begins by reporting our observations and findings in using MiniOS at the University of Northern British Columbia (UNBC). It elaborates on the use of MiniOS as a prototyping and research platform. Then, it presents student feedback, and finishes by discussing further research concerning the experimental evaluation of the reported observations, as well as the unanswered objective mentioned above. The analysis is mostly based on anecdotal evidence and hence qualitative in nature. We do not have enough data to make a quantitative analysis that have statistical significance.

5.1 Observations and Findings

Reflecting on the experience to date, MiniOS has served well as an instructional system. Previous versions of the teaching platform (Figure 5.1) were used as the laboratory component for OS (CPSC 321) in Fall 2013, 2014 and 2015, and as the laboratory component for computer architecture (CPSC 231) in Winter 2015 and 2016.



(a) 2013 (b) 2014 (c) 2015

Figure 5.1: MiniOS architecture in different years

Figures 5.1(a), 5.1(b), and 5.1(c) illustrate the evolution of MiniOS. Overall the

delivery of the material went without problems, and in the process we gathered experiences.

5.1.1 Book experience

The role of the book seemed to have served its purpose. The amount of details offered in its latest version allowed for students to complete assignments (both CPSC 231 and 321) with little necessity to rely on other sources. In early offerings where the guide covered less material, students consistently indicated being frustrated for the lack of related external sources. Moreover, the amount of details seemed to have enabled students to complete projects. In Fall 2013, CPSC 321, two of the five teams using the SAM4S board (two teams used different hardware) were not able to present working projects due problems of technical nature. In 2014, the number went down to one and that team used different hardware. In 2015, it went down to zero, and all teams used the SAM4S board.

The system part of the guide assumes that students have had some experience working with the SAM4S board and Cortex-M architecture. When we delivered CPSC 321 in 2015 this was not the case for every student, as some had taken CPSC 231 one year earlier (in 2014) using a different CPU architecture. Interestingly, the lack of previous Cortex-M experience did not seem to matter considerably. Three of the thirteen students attending labs did not have previous Cortex-M experience. Still, based on marks and personal appreciation, they performed similar to the rest of the class. In fact, one of them went to obtain the highest marks in labs. It is quite possible and reasonable that they dedicated more time to get on track with the new hardware platform and programming environment.

5.1.2 Instruction and tutorial experience

Based on previous experience teaching labs by building an OS for x86, MCU embedded hardware seemed to have allowed us to dedicate less instruction to present students with hardware details. Specially with the use of the guide, the required concepts previously introduced in CPSC 231 were just referenced and not re-introduced.

Tutorials were offered in a classroom once a week. Despite the material being covered in the guide, many times students needed further clarification. While some were able to finish labs with minimal or no consultation at all, others did not. Thus, it is recommended to have dedicated lab or tutorial sessions, where the lab instructor gives an oral presentation of the material. To gain insight into what is difficult and what is not, and what could use extra instruction, it is advised that the lab instructor solves the labs in advance.

5.1.3 Language experience

Java is the language used to teach most courses at UNBC. This means that, for many students, CPSC 321 was their first encounter with C language. Among all the C-specific concepts, pointers and pointers to pointers demonstrated to be difficult to decipher. In fact, students consistently reported much of the struggle with assignments came from inexperience with the C programming language. To compensate for it, tutorials covered the use of pointers, callbacks, structures, organization of code in modules, use of header files, compiler attributes, among other relevant C specifics. Often students were not able to proceed further due to a specific language detail they were confused about or not knowledgeable of. While some students were prompt to

ask, other were not. Those that did not ask reported spending a considerable amount of extra hours figuring out by themselves. Thus, students were encouraged to ask for language related doubts. In general, it is recommended that the lab instructor does not hesitate in assisting students with language problems.

5.1.4 Debugger experience

The presence of debugging facilities, albeit circumstantial, showed to be very important for solving the laboratory assignments. On occasions, the debugger was the difference between students completing an assignment or not. Often assistance was given in the form of debugging sessions. Sometimes because the lab instructor was unsure where the mistake was, and some other times because the debugger allowed for a demonstration of a concept that was otherwise not being understood from an oral explanation. Also, we have found that most of students' bugs are due not to wrong logic (they usually get it right), but due to a missing technical detail or a wrong assumption of technical nature. Debugging was very useful in finding those mistakes, as it enabled to verify step by step the details and assumptions of what is supposed to happen versus what is actually happening. This contrasts with typical remote debugging, which is a rather limited way of debugging (similar views are expressed by Holland et al[18]).

We also noted that in spite of previous debugging experience, in most cases, students lacked the debugging abilities to take advantage of the facilities available. In this regard, videos showing effective use of the debugger, as well as personal assistance, were provided. Interestingly, students seemed to have improved their bug-finding skills after only a couple short sessions of personal debugging assistance.

5.1.5 Hardware experience

In the first offering of the course, the MCU platform had a neutral reception. This, however, has changed for the later two offerings of the course. Perhaps the guide played a role in that. For the latest offerings of both CPSC 231 and CPSC 321 more than half of students showed interest and enthusiasm of working with hardware.

Overall, boards behave well. On occasions, albeit not often, boards would simply fail to be recognized by Atmel Studio. Some times resetting the host computer or replugging the board would fix the problem, but other times the board would continue to fail and a replacement had to be given. So it is recommended to have extra boards in case they are needed.

Working with external peripherals can sometimes be a problem. There was a few incidents where Xbee modules and one board where burnt due to wrong wiring. Being computer science majors, a good number of students showed problems with wiring of external peripherals. For example, late in the course one student started having problems integrating an Xbee for his final project, and expressed that CPSC 321 was (until the issues started) his favourite course in that semester. These problems repeated in a few occasions, and as a consequence, the latest version of MiniOS has stopped requiring any use of peripherals that are not expansion boards, since they simply plug into an expansion slot. Concretely, Xbee radio modules have been replaced by the REB233 board. At the time adding external support for PS/2 keyboards was also being considered, but had to be dismissed for the same reasons.

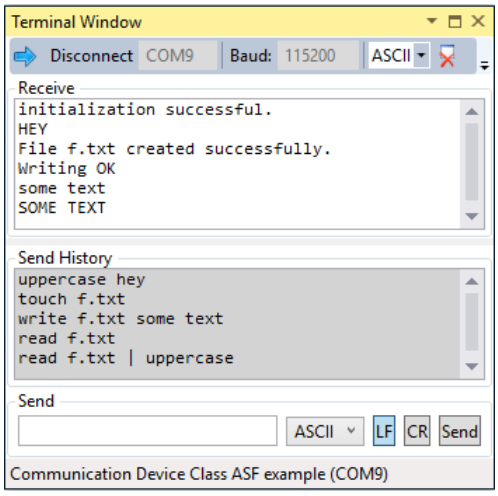
For final projects, students choose something of their interest, and often they require external hardware that requires wiring. In this regard, not only wiring, but

finding the right piece of hardware have been consistently mentioned (in project reports) to be challenging. For our part, we have provided assistance in both.

5.1.6 Project experience

At the end of the semester we held, for both courses, a final project presentation, where students had to give a short presentation and demonstration of their projects. Figure 5.2 shows two sample projects. Figure 5.2a shows a clock alarm project that runs MiniOS. Figure 5.2b shows a project named *Pinto pipes*, a rudimentary command line interpreter that supports redirection. Other projects include *Ascii at a distance* (a communication API for wireless devices), *SOS (simple operating system)*, *Remote sensor drone* (a remote rover with sensing capabilities), thread signalling for MiniOS, and *System Security* (secure user account management).

(a) Clock Alarm Project



(b) Pinto Pipes Projects

Figure 5.2: Sample student projectss

5.1.7 Drivers experience

An important aspect of MiniOS is the integration of third-party open source firmware. This enabled seamless integration of a variety of different peripherals. With available drivers, the adding of hardware functionality to the system became a mechanical task. We found this to be good for student engagement, as driver availability is the main limitation in using external hardware in projects. In fact, during project proposals we advised students to check for driver availability before acquiring any peripheral. It is worth noting that by themselves, drivers are of little use as their use is difficult to figure out from documentation. The device driver guide played an important role in simplifying it, and turning it into a mechanical task.

5.2 MiniOS as a prototyping research platform

The amount of functionality built-in, together with the ease of hardware integration, made MiniOS a good alternative for embedded systems prototyping. In particular, applications have access to OS facilities, as well as straightforward sensor and actuator integration. Prototyping platforms with similar capabilities are Microsoft's .NET gadgeteer[35] and mbed [2]. Figure 5.3 shows two mobile robots part of a experimental multi-robot platform based on MiniOS.

Likewise, given that MiniOS' inner working are well documented and are comparatively simpler that other systems, it could serve well as systems research platform where system designs can be tried in relatively small time frame. Up to this point, we have discussed our appreciation of the teaching experience. The next section discusses feedback received from students.

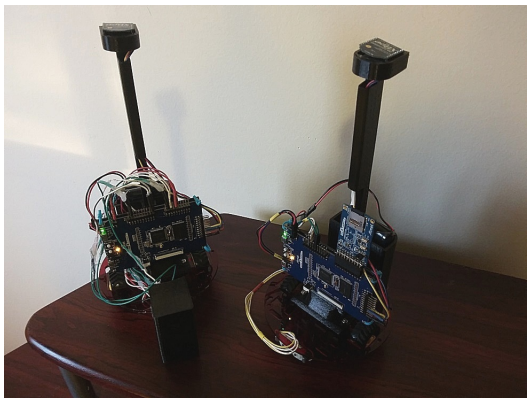


Figure 5.3: MiniOS-based multi-robot platform

5.3 Students' feedback

In Fall 2014, students were asked to elaborate in what they thought were strengths and weaknesses of laboratories projects. These are some of the answers.

S1: Strengths: The assignments are very hands on and we get to see the things we discuss in class. The example programs show the functionality of the board.

Weaknesses: There seems to be little documentation for the ASF. The coding can be hard to follow

S2: Assignments are a good way to see the complexity and challenge in dealing with the hardware level. They are nice in that you can access the hardware directly, and use the debugger that is provided to actually "see" the registers and the hex or binary values stored here, and how things are interacting. However, that is also its main weakness. being that they are reasonable complex pieces of software it is very challenging to understand how all the components are interacting at times. [...] Along with the large amount of digging that needs to be done to understand the documentation, the other challenge is understanding

C as I have not used it much to although things are similar it is still not Java. however it is kinda nice [illegible] to use something other than Java.

S3: In my opinion, the programming assignments are a mixed bag. I think its a good way to show incrementally how each part of an OS is designed and implemented in practice. The assignment themselves strengthen the knowledge learned in class.

The downside, however, is the language implemented. Its a minor point, but it is an issue with which I struggle. Until this course, I've never used/been/seen exposed to C. It just makes understanding and implementing ideas needlessly complex.

S4: [...]It is too easy to get stuck on a simple task specially when the student is using a new language and development environment that is foreign.

Being a Java university, the first assignment should a strictly C assignment. Designed to get an understanding of the differences from Java, and features required to use the Atmel libraries. This can be assigned day 1 of the term.

S5: Strengths: Got to see and develop an entire OS. Get to use a simple enough platform to feasibly develop all components. The interactive nature of the platform makes progress easy to see and rewarding. Tutorials are well written and provide detailed instructions. Code base is quite small and it is easy to hold entire system in your head.

Weaknesses: Lack of documentation and online support for platform. Each assignment is very involved and requires a large time commitment.

S6: Strengths:

- The projects are fun, engaging and rewarding

- Interacting with real hardware is great
- The resources provided by the TA are complete and helpful

Weaknesses:

- C is not something I am particularly familiar with
- Atmel resources (documentation) are not always helpful
- C language guides are not always applicable

For fall 2015 substantial changes were made to the guide. In particular:

- It was made more self-contained, and this it made little or no reference to external documentation.
- Additional missed necessary details dealing with the architecture were covered.
- Tutorial time was devoted for looking at specific C knowledge required to complete assignments.
- Certain embedded systems specific details not relevant to operating systems was removed.

Students were again asked to elaborate in what they thought were strengths and weaknesses of laboratories projects. These are some of the answers (here we group them)

- Strengths:
  - Physical depiction of what we're doing (interactive buttons & oled screen)

- Overall great assignment layout. I don't understand why people need extensions!
- Everyone loves bonus questions!
- Nice to have many examples/viewpoints
- I've heard from many that they're having trouble with their board. Not me personally though.
- Great examples to draw from.
- Much prior use of board/software
- enforces understanding of:
  * Interrupt
  * HAL
  * better code
  * better structure
- FUN!
- The documentation provided is top notch. It makes the world of a difference having lab documents and driver documents written in PLAIN LANGUAGE which speeds up learning.
- Unified system. The SAM4S is easy to work with, we have been using it for a few years, so students aren't totally new to developing for it.
- Software support. It's undeniable Atmel Studio is useful in learning how to code for embedded systems. Visually stepping through code and viewing memory being modified helps intuition. As well as debugger.
- Relevant work, simplistic design. it's easy to develop on ARM and learn the ropes. Jumping to x86 would be more difficult. ARM is also very popular and won't be going away soon.

  - Assignments are relevant to course. It's easier to break down OS concepts and learn how to code them [illegible]. it solidifies the ideas and makes classes easier to understand.

- Weaknesses:

  - lack of any useful documentation about the SAM4S online
  - C is a bit tough when no taught any prior C (pointers, memory is odd) (still not so hard)
  - Atmel Studio 6.0, 6.2 a bit finicky and error prone (better with 7.0 now!)
  - **For me* Many others would disagree:
    * not so hard enough sometimes
    * would like to build some driver from scratch (camera, touchscree, etc)
  - Assignment are long. A lot of time is required to complete. Due to bugs it can sometimes take more than a week. Two weeks is usually required.
  - Atmel Studio is buggy, it leads to a lot of wasted time messing around.
  - It can sometimes be difficult to tie into classroom lectures. It would almost help to focus lecture on how ARM systems can have an OS build on them to have more insight.

Although our evaluation is purely qualitative, we would like to draw more general conclusions from it. In the next section, we discuss the possibility of doing so.

5.4 Discussion

Arguably, the smaller a system is, the better it lends itself for construction or modification. Similarly, the less sophisticated an architecture, and the more instructional material is provided for it, the less effort that must go into its comprehension. In that sense, we believe, this thesis work is justified. In fact, it was derived precisely from that rationale. However, we do not know whether the MiniOS platform fulfils its objective of teaching operating systems principles. We would like not only to know this, but also to explore the possibility of generalizing some of the observations and findings collected while using MiniOS to teach operating systems laboratories. We dedicate this section to discuss these possibilities, as well as to provide some background on experimental computing education research.

5.4.1 Experimental research in computer science education

When proposing or experimenting with a new teaching tool or approach, the obligatory question is—does it work? If the answer is yes, then the follow up question is: is there evidence that it works? A review of computer science education publications will quickly reveal the answer to this question. For a majority of proposed educational approaches and tools, the answer is no. As Lister criticizes in [22] and [23], solutions to problems in computing education research must be validated by evidence, not intuition and introspection. On this same topic, Guzdial [15] states:

> "Without evidence, teachers rely on intuition informed by experience. Sometimes that intuition may be informed by years of experience. Sometimes that experience is not at all relevant."

Practically, it can also be a problem. Consider the concerns raised by Fincher [13] regarding the problems that secondary school teachers face when adopting teaching techniques and tools:

> "We need a program of educational research to support teachers, to ensure ideas work in real classrooms and with real teachers—and so we do not repeat cycles of error. Teachers are faced with a plethora of plausible approaches and no way to choose between them but the conviction (and charisma) of their inventors [...] [The] evidence these are based on is solely "Do it like this! It works for me!"

Does this mean computer science education research without experimentation has no value at all? Valentine [34] argues that, in spite of the lack of experimental data, solutions to problems are valuable contributions. Still, if we are to do computing education research, eventually we ought to grow as a community and take that extra step of validating our solutions with evidence.

Likewise, Hazzan[17] claims qualitative research does have its place in computer science education:

> "The nature of quantitative research does not [...] enable the researcher to explore all aspects of complex situations. [...] Qualitative research approach enables us to highlight many angles of people-centered situations."

Then, he goes on to say:

> "It is suitable to employ a qualitative research approach mainly in the study of personal experiences and processes (such as learning, understand-

> ing, teaching, choosing), which are descriptive in nature. Accordingly, and naturally, it would be appropriate to study such topics based on the analysis of verbal-descriptive data."

The approach suggested by Hazzan is to begin with exploratory qualitative research as to collect data in the form of observations and interviews; then, in a second quantitative stage, test specific hypotheses that are drawn from these observations and interviews. Lastly, based on the findings of the second sage, do a third qualitative research with the purpose of gaining new perspectives on the original results, and even perhaps explain them. In either case, it is clear that research in computing education is going in a direction where experimental research goes hand in hand with qualitative research.

Part of the work presented in this thesis is equivalent to the first step described above. It presents not only the MiniOS platform per se, but also observations and feedback in using it as instructional laboratory material. For future research we would like to generalize some of the results of our work by engaging in the second and third stages. Moreover, we would like to give definitive answer to the question concerning MiniOS instructional value.

5.5 Summary

This chapter presented our observations and findings while using MiniOS to teach laboratory projects at the University of Northern British Columbia from 2013 to 2016 on different perspectives; those dealing with the book, instruction, tutorial, language, debugger, hardware, projects, and drivers. Then, the use of MiniOS as prototyping

research platform was briefly considered, and student feedback was presented. Finally, we discussed experimental research in computing education and considered the qualitative results of this work as part of a larger research endeavour.

Chapter 6

Concluding Remarks

6.1 Conclusions

The main contribution of this thesis is MiniOS, an instructional platform for the delivery of operating systems laboratories. MiniOS follows on the steps of instructional systems that attempt to deal with complexity by lowering the code volume. We go further and identify the target hardware platform as an additional source of complexity that can also be account for. The result is a MCU instructional operating system, and to our knowledge, the first one of its type. In addition, the platform includes a step-by-step guide to its construction whose purpose is to offer all the necessary details for the construction of the system.

The platform was used in three different occasions to deliver laboratory assignments for an introductory course in operating systems. Student feedback was overall favourable. We presented this feedback together with other experiences.

MiniOS cannot—and is not intended to—replace more traditional desktop-centered design approaches; instead it serves as an alternative. For example, if a course has as objective to provide students with experience in topics related to the MMU (e.g. virtual memory), or to teach OSes as they are built in industry, the MiniOS approach

is a poor fit.

University laboratories are not the only place where MiniOS has found use. Given the amount of built-in functionality and the ease with which hardware can be integrated, it can, and has, been used as a rapid embedded prototyping platform. Likewise, being well-documented, it serves as a systems research test bed. Currently the only existing ports is the SAM4S Xplained Pro board, but there is no reason that would not allow MiniOS to be ported to other MCU platforms. In fact, it was designed to be ported.

6.2 Future Work

For future work, we would like to explore the possibility of porting the instructional platform to other MCU platforms like Arduino Zero, a Cortex-M0 based Arduino. The integration of bare-metal drivers as part of MiniOS might work well with the plethora of available Arduino code online. Since there is existing infrastructure for ad-hoc wireless connectivity, it is possible to add internet connectivity and customize MiniOS to work as an IoT OS.

The guide, together with the SAM4S board have been used in two occasions to deliver computer architecture labs. These results, however, have not been published, and many have not been discussed in this work. Thus there is material for future work. The samples of the book presented in this thesis are the 2015 version. Currently we are working on a newer version that we intend to publish like a small book or e-book. Finally, there is space for further work on generalization of some of the findings and observations made during the use of the platform in the past three years.

Appendices

Appendix 1

(excerpt from scheduler chapter)

Threads

```
typedef struct{
     uint8_t* name;
     uint32_t* sp;
     ThreadState state;
}MiniThread;
```

Figure 4.9

The concept of thread is abstract and can mean slightly different things on different contexts. As far as we are concerned, a thread is an instance of MiniThread (Figure 4.9). Threads have a name, a stack pointer, and, at all times, they are in one of more possible states. (Do not confuse state as discussed in this subsection, with state context as discussed in the previous subsection.) So far a thread's state can be either *ready* or *running*, and we represent this in the form of state diagrams (Figure 4.10).

From an application perspective, applications should be allowed to create threads via a system call; say, *thread_create*. Following other thread library interfaces, we arbitrarily make the *thread_create* system call to take (in this order): a pointer to the thread's code, its name as a string, and the size of its stack in 32-bit elements (not bytes). (Figure 4.11 shows an example of its use to create a thread.)

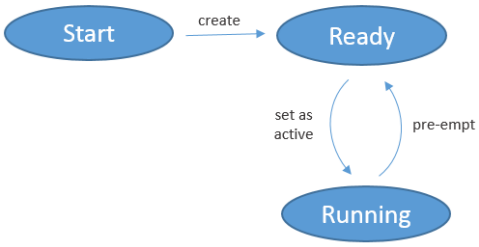


Figure 4.10

```
void t1(void){ ... }

int main(void){
     thread_create( t1, "thread 1", 512 );
}
```

Figure 4.11

Now, we expand our non-threaded scheduler to support one thread; and we approach this by attempting to write a definition for *scheduler_thread_create (*the implementation of *thread_create* within the scheduler).

One-thread scheduler Part One

```
...
MiniThread thread;

void scheduler_thread_create( void (*code)(void),
                              uint8_t* name,
                              uint32_t stack_sz ){
          thread.name = name;
          thread.state = ThreadStateReady;
          thread.sp = ...
}
```

Figure 4.12

Starting with the obvious, we write the incomplete definition in Figure 4.12, which leads us to an important question—what is the initial SP for a new thread? Clearly it must be an address within the process stack. Since the Cortex-M4 stack grow downwards, it must be the **end of the stack** for the thread being created. This in turn raises a second question—where is the stack for some given thread? So far nowhere. We have not allocated any stack space for any thread. Then a third question follows—on creation, how do we allocate space for a thread's stack? Now we address the problem of allocating stack space for a thread; and, although there are different ways this can be done, we do it very in the simplest way the author was able to imagine: **one after another**.

One-thread scheduler Part two (stack allocation for threads or y los detalles siguen)

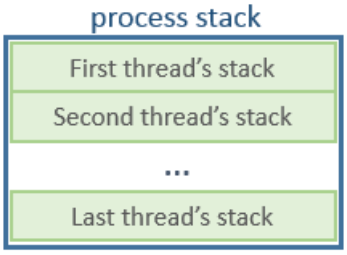


Figure 4.13

Let the end of the process stack be *epstack*. Initially, before creation of the first thread, the process stack is empty. So the SP for the first thread is *epstack*. Following a *"one-after-another"* layout, SP for the second thread must be *epstack - the first thread's stack size in bytes;* and so forth

until no more space is available. This is depicted in Figure 4.13. (Note how **allocating space is as simple as setting the right SP for every thread, on creation**.)

```
...
MiniThread thread;

void scheduler_init(void){
     stack_init( hal_cpu_get_epstack() );
     ...
}

void scheduler_thread_create( void (*code)(void),
                              uint8_t* name,
                              uint32_t stack_sz
                            ){
     thread.name = name;
     thread.state = ThreadStateReady;
     thread.sp = stack_get(); //set SP

     stack_alloc( stack_sz ); //space for
                              //thread's stack
}
```

Figure 4.14

The corresponding code is in Figure 4.14. *stack_get* returns a pointer to the top of the allocated stack so far, while *stack_alloc* allocates space simply by moving down the pointer. In this case the SP is moved down by *stack_sz* 32-bit elements. (The exact implementation details are left as exercise.) Once SP has been set correctly, when a thread executes for the first time it will do it starting on its own stack.

Everything seems to be in place now. Memory is allocated for a thread and its SP is set accordingly on thread creation. Then, on context switch, the pre-empted thread's context is saved and the new active thread's context is restored. This takes us to consider yet another detail: for a thread to be pre-empted, it must first be in execution. Hence, now we look at the problem of execution of the first thread. In particular, creation and execution of main—yes, main is also a thread!

```
void Reset_Handler(void){
   ...
   //Initializes the system
   system_init();

   //initializes and starts the scheduler
   scheduler_init();

   //set CPU in user mode
   hal_cpu_set_psp( hal_cpu_get_epstack() );
   hal_cpu_psp_active();
   hal_cpu_set_unprivileged();

   //Creates and starts main thread
   thread_create( main, "main thread", 512 );

   //Execution will never reach here
   while (1);
}
```

Figure 4.15

One-thread scheduler Part three (creation and execution of the main thread or los detalles no tienen fin)

Although we could do this in various different manners, we write *scheduler_thread* in a way that, when called for the first time, instead of returning back as normal, it transfer execution to the newly created thread. Specifically, as it shown in Figure 4.15. (Two things must be noted here. First, we are invoking a *thread_create* system call; this is because at that point the machine is in user mode. Second, execution will never return from *thread_create* back to the reset handler.)

To see how to transfer execution from within *scheduler_thread_create* to the newly created main thread, consider the following—how exactly does execution gets to *scheduler_thread_create*? It gets there **from user mode via an SVC call** i.e. an exception. How does the CPU know where to go back on exception return? There is a stacked PC that was placed there on exception entry. What would happen if we replace the stacked PC with the address of main? Then execution would transfer to main—and that is exactly what we do. **We insert a custom hardware context, thereby simulating that execution was in main before exception entry**. This is depicted in Figures 4.16. (Note SP must too be set accordingly for the unstacking to go the way we want.)

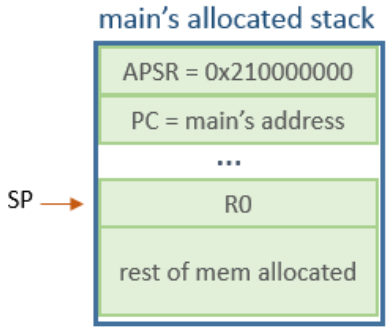


Figure 4.16

(excerpt from stack chapter)

Stack

A stack is an abstract data type with two operations: push and pop. Push adds one element to the stack, pop removes one from it. Operations are performed in a way that, given a stack in state *S*, a push operation followed by a pop operation will leave the stack in the same state *S*.

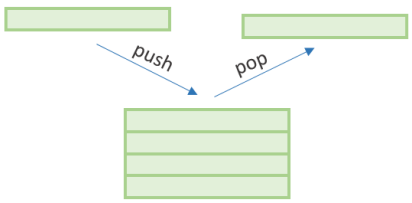


Figure 5.0

Visually, push adds an element to the top of the stack, and pop removes an element also from the top (Figure 5.0). Data is thus added and removed in a last-in-first-out (LIFO) manner.

Except, perhaps, for a research prototype, all Von Neumann machines have built-in support for one or more stacks. (To relate this with previous knowledge on data structures, think of how a stack can be implemented as a linked list, or an array, or even two queues. In the machine's case it is implemented as raw memory bytes.)

Implementation in hardware

Like any abstract concept, there is gap between its theory and implementation. In the case of a hardware stack this is a considerable gap; and one has to be careful, when reasoning about it, to keep the implementation details in mind. Specifically, **a stack as such does not exist in the architecture.** We limit ourselves to utilize the facilities provided to "pretend"[1] there is one.

Facilities vary among architectures, but they typically include a stack pointer register, and special instructions to push and pop element in and out.

Cortex-M4 Stack facilities

The **stack pointer register (SP)** is a special purpose register pointing to the top of the stack. Importantly, even though, visually we think of the stack as growing upwards (as a pile of plates), in the Cortex-M4, like in other architectures, the stack grows downwards. In other words, the more data the stack has, the lower the memory address the stack pointer holds. This is depicted in Figure 5.1.

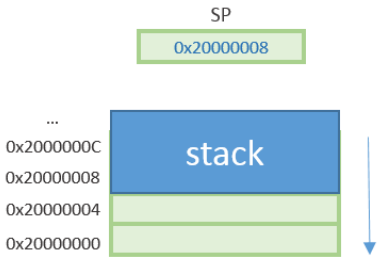


Figure 5.1 – Stack grows downwards

(Keeping this *confusing-non-intuitive-against-gravity* way of visualizing the stack let us continue.)

A **push instruction,** with syntax *push {reg}*[2]*,* moves SP down one word, then stores the register *reg* to the memory location pointed by SP. Contrariwise, a **pop instruction**, with syntax *pop {reg}*[3], loads *reg* with the value stored in the memory address pointed by SP, then it moves SP one word up (effectively "removing" that element from the stack).

[1] One could argue all abstractions are pretensions. For instance, we pretend a link list is a stack when using the first one when implementing the second one. The difference I see is: in the case of the machine's stack, memory locations are never completely abstracted as stack elements, as general memory instructions still can be applied to them. Therefore, knowledge of both memory, and architecture, as well as imagination are neccesary. The closest example I can think of is that of weak typing in programming languages. You may have, for example, a char type in C, but that doesn't prevent you from doing integer arithmetic with a char, thus requiring extra knowledge of how characters are encoded as integer values. If you're forced to think of integers when dealing with characters, then such a "char abstraction" is rather loose.

Appendix 2

(excerpt from device driver's guide)

SSD1306 OLED Display

Description

The display in the OLED1 extension board is a 128x32 pixel white monochrome OLED Display. It is driven by a *SSD1306 display controller* from Solomon Systech. It interfaces with the MCU via SPI (serial peripheral interface).

ASF Modules required

- SSD1306 OLED Controller (component)

Demo

The demo prints a message on the screen. See SSD1306 OLED Demo.

Notice only text of pre-defined size is supported. However it is possible to drive the display to display more things. This is an example. This is another example. Maybe you feel like modifying the drivers to allow for bigger fonts or even shapes.

Demo Code

```
#include <asf.h>
#include <string.h>

int main(void)
{

	sysclk_init();
	board_init();

	// Initialize SPI and SSD1306 controller.
	ssd1306_init();

	// Clear screen.
	ssd1306_clear();

	//Set line and column to 0
	ssd1306_set_page_address(0);
	ssd1306_set_column_address(0);

	/// -------- First Screen --------
	ssd1306_write_text("Coffee consumption improves");

	delay_ms(1500);

	ssd1306_set_page_address(1);
	ssd1306_set_column_address(8);

	ssd1306_write_text("programming performance");

	delay_ms(1500);

	ssd1306_set_page_address(2);
	ssd1306_set_column_address(20);

	ssd1306_write_text("when coding in C.");

	delay_ms(1500);


	/// -------- Second Screen --------
```

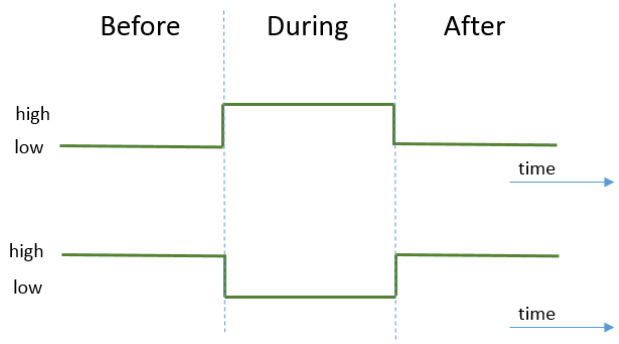
```
        ssd1306_clear();
        ssd1306_set_page_address(0);
        ssd1306_set_column_address(0);

        uint8_t text[65];
        uint8_t* pText = text;
        uint8_t *char_ptr;
        uint8_t i=0, column=0, page=0;

        //use sprintf to create strings from numbers, variables and other strings
        sprintf(text, "When coding in Java, however, performance decreases in %f %%", 73.37);

        //print text character by character
        while(pText){
                //write a single character
                char_ptr = font_table[*pText++ - 32];
                for (i = 1; i <= char_ptr[0]; i++) {
                        ssd1306_write_data(char_ptr[i]);
                }

                //newline
                if(column++ == 35){
                        column = 0;
                        page++;
                        ssd1306_set_column_address(column);
                        ssd1306_set_page_address(page);
                }

                //wait
                delay_ms(100);
        }


}
```

(excerpt from device driver's guide)

Buttons

Description

Unlike other Parallel IO-based devices, mechanical buttons deserve their own entry. This is because they are peculiar: when pressed, they bounce. We like to think that when a button is pressed it will change the IO line's state and when the button is released its state will go back. Something like this:



The physical world is never that ideal, however. When a mechanical button is pressed it bounces, therefore generating a train of pulses instead of just one.

This is then interpreted as the button being pushed several times. The ATSAM4SDC32 has hardware support for de-bouncing, which allows to filter pulses which duration is less than a specified threshold. This will not eliminate all the glitches, but it will make it much better; so expect a few of them when you press buttons. Another way is to do it by software, but this requires intervention of the CPU. The idea is the same, whenever a change in state is detected in an IO line, check the IO line again a few milliseconds later; if the state is the same then the button was pressed, else it was a glitch. Maybe even check the IO line in several occasions after the first pulse and determine that the button was pressed only when the state of the button was the same in all the occasions (an example of this is shown in one of the Parallel IO entries… the one with the movement sensor).

This video explains further. This is a software de-bouncing example in Arduino.

ASF Modules required

- Same as Parallel IO

Demo

The Button Demo toggles LEDs when buttons are pressed. The drivers allow to set a “cut-off frequency for the de-bouncing filter” as the last parameter of *pio_set_debounce_filter*.

See Buttons Demo.

Demo Code

```
#include <asf.h>

#define LED1  IOPORT_CREATE_PIN(PIOC, 20);
#define LED2  IOPORT_CREATE_PIN(PIOA, 16);
#define LED3  IOPORT_CREATE_PIN(PIOC, 22);

void Button_Handler(uint32_t id, uint32_t mask)
{
	uint32_t led;

	if ( ID_PIOA == id && PIO_PA0 == mask ){ led = LED1; }
	else if( ID_PIOC == id && PIO_PC29 == mask ){ led = LED2; }
	else if( ID_PIOC == id && PIO_PC30 == mask ){ led = LED3; }
	else { return; }

	ioport_set_pin_level( led, !ioport_get_pin_level(led) );
}

const uint32_t irq_priority = 5;
void configure_buttons(void)
{
	//Configure Pushbutton 1
	pmc_enable_periph_clk(ID_PIOA);
	pio_set_debounce_filter(PIOA, PIN_PUSHBUTTON_1_MASK, 10);
	pio_handler_set(PIOA, ID_PIOA,
	PIN_PUSHBUTTON_1_MASK, PIN_PUSHBUTTON_1_ATTR, Button_Handler);
	NVIC_EnableIRQ((IRQn_Type) ID_PIOA);
	pio_handler_set_priority(PIOA, (IRQn_Type) ID_PIOA, irq_priority);
	pio_enable_interrupt(PIOA, PIN_PUSHBUTTON_1_MASK);

	//Configure Pushbutton 2
	pmc_enable_periph_clk(ID_PIOC);
	pio_set_debounce_filter(PIOC, PIN_PUSHBUTTON_2_MASK, 10);
	pio_handler_set(PIOC, ID_PIOC,
	PIN_PUSHBUTTON_2_MASK, PIN_PUSHBUTTON_2_ATTR, Button_Handler);
	NVIC_EnableIRQ((IRQn_Type) ID_PIOC);
	pio_handler_set_priority(PIOC, (IRQn_Type) ID_PIOC, irq_priority);
	pio_enable_interrupt(PIOC, PIN_PUSHBUTTON_2_MASK);

	//Configure Pushbutton 3
	pmc_enable_periph_clk(ID_PIOC);
	pio_set_debounce_filter(PIOC, PIN_PUSHBUTTON_3_MASK, 10);
	pio_handler_set(PIOC, ID_PIOC,
	PIN_PUSHBUTTON_3_MASK, PIN_PUSHBUTTON_3_ATTR, Button_Handler);
	NVIC_EnableIRQ((IRQn_Type) ID_PIOC);
	pio_handler_set_priority(PIOC, (IRQn_Type) ID_PIOC, irq_priority);
	pio_enable_interrupt(PIOC, PIN_PUSHBUTTON_3_MASK);
}

int main(void){
	sysclk_init();
	board_init();
	configure_buttons();

	while(1);

}
```

Bibliography

[1] Charles L. Anderson and Minh Nguyen. A survey of contemporary instructional operating systems for use in undergraduate courses. *J. Comput. Sci. Coll.*, 21(1):183–190, October 2005.

[2] ARM. mbed, 2016.

[3] Benjamin Atkin and Emin Gün Sirer. Portos: An educational operating system for the post-pc environment. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '02, pages 116–120, New York, NY, USA, 2002. ACM.

[4] Mordechai (Moti) Ben-Ari. In defense of programming. *ACM Inroads*, 7(1):44–46, February 2016.

[5] Michael D. Black. Build an operating system from scratch: A project for an introductory operating systems course. *SIGCSE Bull.*, 41(1):448–452, March 2009.

[6] Alex Chadwick. Baking pi: Operating systems development, 2012.

[7] ChaN. Fatfs - generic fat file system module, Unknown.

[8] Wayne A. Christopher, Steven J. Procter, and Thomas E. Anderson. The nachos instructional operating system. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pages 4–4, Berkeley, CA, USA, 1993. USENIX Association.

[9] Douglas Comer. *Operating System Design: The XINU Approach.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[10] Marc L. Corliss and Marcela Melara. Vireos: An integrated, bottom-up, educational operating systems project with fpga support. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 39–44, New York, NY, USA, 2011. ACM.

[11] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, October 1972.

[12] George Fankhauser, Christian Conrad, Eckart Zitzler, and Bernhard Plattner. *Topsy – A Teachable Operating System*, April 2000.

[13] Sally Fincher. What are we doing when we teach computing in schools? *Commun. ACM*, 58(5):24–26, April 2015.

[14] Michael Goldweber, Renzo Davoli, and Mauro Morsiani. The kaya os project and the umps hardware emulator. *SIGCSE Bull.*, 37(3):49–53, June 2005.

[15] Mark Guzdial. Bringing evidence-based education to cs. *Commun. ACM*, 58(6):10–11, May 2015.

[16] Mark Guzdial. What's the best way to teach computer science to beginners? *Commun. ACM*, 58(2):12–13, January 2015.

[17] Orit Hazzan, Yael Dubinsky, Larisa Eidelman, Victoria Sakhnini, and Mariana Teif. Qualitative research in computer science education. *SIGCSE Bull.*, 38(1):408–412, March 2006.

[18] David A. Holland, Ada T. Lim, and Margo I. Seltzer. A new instructional operating system. In *Proceedings of the 33rd SIGCSE Technical Symposium on*

Computer Science Education, SIGCSE '02, pages 111–115, New York, NY, USA, 2002. ACM.

[19] David Hovemeyer, Jeffrey K. Hollingsworth, and Bobby Bhattacharjee. Running on the bare metal with geekos. *SIGCSE Bull.*, 36(1):315–319, March 2004.

[20] Digi International Inc. Connect devices to the cloud, 2016.

[21] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[22] Raymond Lister. Rare research: Why is research uncommon in the computing education universe? *ACM Inroads*, 3(4):16–17, December 2012.

[23] Raymond Lister. Teaching-oriented faculty and computing education research. *ACM Inroads*, 3(1):22–23, March 2012.

[24] Haifeng Liu, Xianglan Chen, and Yuchang Gong. Babyos: A fresh start. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '07, pages 566–570, New York, NY, USA, 2007. ACM.

[25] Real Time Engineers Ltd. Freertos, 2016.

[26] Ben Moseley and Peter Marks. Out of the tar pit. In *SOFTWARE PRACTICE ADVANCEMENT (SPA)*, 2006.

[27] Ben Pfaff, Anthony Romano, and Godmar Back. The pintos instructional operating system kernel. In *Proceedings of the 40th ACM Technical Symposium on*

Computer Science Education, SIGCSE '09, pages 453–457, New York, NY, USA, 2009. ACM.

[28] Rene S. Pinto, Pedro Nobile, Edwin Mamani, Loureno P. Junior, Helder J.F. Luz, and Francisco J. Monaco. Operating system from the scratch: A problem-based learning approach for the emerging demands on {OS} development. *Procedia Computer Science*, 18(0):2472 – 2481, 2013. 2013 International Conference on Computational Science.

[29] Christopher Svec. Freertos @ONLINE, 2016.

[30] Andrew S. TanenBaum and Albert S. Woodhull. *Operating System Design and Implementation*. Pearson, third edition, 2006.

[31] Jonathan Tang. Write yourself a scheme in 48 hours, 2015.

[32] Unknown. *Embedded Xinu*, September 2013.

[33] Unknown. Xda-developers:android, January 2015.

[34] David W. Valentine. CS educational research: A meta-analysis of sigcse technical symposium proceedings. *SIGCSE Bull.*, 36(1):255–259, March 2004.

[35] Nicolas Villar, James Scott, Steve Hodges, Kerry Hammil, and Colin Miller. .net gadgeteer: A platform for custom devices. In *Proceedings of the 10th International Conference on Pervasive Computing*, Pervasive'12, pages 216–233, Berlin, Heidelberg, 2012. Springer-Verlag.