Building a Music Player: Teaching Operating Systems Concepts via an Arduino-Powered Music Player

John Seng
Dept. of Computer Science
California Polytechnic State University, San Luis Obispo
San Luis Obispo, CA 93407
jseng@calpoly.edu

***Abstract*—The sequence of assignments outlined in this work targets university level computer science and computer engineering students who are familiar with C and some form of assembly. First, the students build a preemptive mini-operating system with support for multiple thread contexts. Next, the students add in synchronization primitives for sharing values between threads. After the synchronization primitives, the next assignment is to write code to read data from a Linux-compatible filesystem. The final assignment is to put everything together and construct a music player that can play music files stored on a flash memory card. Students also learn about efficient software design when working in a memory-constrained environment such as the Arduino. The end result of the assignments is a music player that students can feel proud of and they fully understand the software behind its operation having built the software essentially from scratch.**

***Index Terms*—operating systems, Arduino, kernel, music, education**

I. Introduction

Lab assignments for teaching operating systems are often taught through a few standard techniques: OS assignments run on virtual machines, running programs on actual hardware, or experiments using simulated OS environments. This paper outlines a set of assignments that use actual hardware. We outline a sequence of assignments that teach operating system concepts by having students develop a mini-OS kernel on an Arduino Mega [1]. The final project for the class is to build a music player that is driven by the kernel.

The Arduino platform is often selected for its ubiquity, ease of use, comprehensive development environment, and readily available libraries. These factors are key to what make the Arduino such a popular platform for educational use. The use of Arduino boards is common in embedded systems courses or even in some introductory programming courses for engineers. In our approach to an operating systems course, we believe that there is much that can be learned by using Arduino hardware, but programming the boards directly using the GCC compiler along with assembly instructions. This development path provides students with hardware that is itself low cost and where the software toolchain is freely available for student use. Using this development process, students build a multithreading kernel from scratch which is then used to control the playback of music files to an output circuit.

This paper is organized as follows: Section II outlines prior work that is related and relevant to this project. Section III outlines the hardware components required to run the assignments. Section IV describes the software architecture and tools used for these assignments. Section V discusses each of the assignments in greater detail. Section VI covers the experiences from running these assignments in a class setting. Section VII describes future work for this project. Section VIII concludes.

II. Related Work

A number of courses use the Arduino to teach embedded systems [11]–[14]. This is the strength of the Arduino platform. Many Arduino projects are programmed as 'bare-metal' projects where the user directly manipulates timers, configures interrupts, and handles coordination among functions. The goal of an Arduino project is often to accomplish some sort of interfacing with a sensor or physical device.

For this class, we are interested in developing an operating system to manage shared resources such as the CPU. There are a number of operating system kernels that run on microcontrollers. Some of these include FreeRTOS [6], ChibiOS [4], ARTE [10], and FemtoOS [5]. Our work is primarily inspired by the ChibiOS operating system, which provides a good model for context switching on a microcontroller platform. This is an open source operating system designed to run on a number of microcontroller architectures with good support for the ARM Cortex [2] embedded microcontrollers.

Upon embarking on creating these assignments, much thought was given to the two most readily available open source hardware platforms for embedded computing: Arduino and Raspberry Pi. Both have their respective advantages. The Raspberry Pi is much more powerful with a full MMU and is capable of running Linux and has been used to teach operating systems courses [15]. An online project outlines the steps in developing a small OS for the Raspberry Pi [3].

In terms of learning for the course, the projects are designed with the belief that students learn greatly by developing things

978-1-5386-1174-6/18/$31.00 ©2018 IEEE

from scratch. Black et al. [9] have a similar goal but with a much more ambitious project.

III. Hardware Overview

The baseline hardware for the assignments described here is the Arduino Mega 2560 R3 [1]. This board is the larger version of the ubiquitous Arduino Uno. Both boards utilize microcontrollers (AVR architecture) and run at 16MHz with the Arduino Mega providing 8KB of RAM. That amount of RAM may seem limiting, but it is sufficient to run the number of threads required for the class. The assignments can be written for the Arduino Uno as well, which has an even more limiting 2KB. Although there are other microcontrollers available with higher clock rates and more memory, we find the capabilities of the Arduino quite sufficient to develop a music player.

Music files are stored on a full size SD card which plugs into a standard Arduino shield. The SD card shield itself plugs into the top of the Arduino Mega and on top of both is an additional breadboard shield. Minimal circuitry is required to listen to audio and that is described in section V-E.

IV. Software Overview

While the Arduino is attractive as a platform because of the readily available Arduino IDE, students develop code using the AVR toolchain which serves as the underlying infrastructure for the Arduino IDE. The toolchain for the Arduino Mega used for these assignments consists of standard open source tools that would be used to develop in C/C++ (e.g. gcc, objdump). We find that the latest version of AVR GCC (7.3.0 in our case) provides good support. Using different versions of GCC may emit code that utilizes registers in differing ways and this is to be avoided.

For the students, supplemental code is provided in terms of struct definitions (for reading the filesystem), accessing hardware functionality, and for accessing the SD card. A large portion of the hardware register details is abstracted away through an API in order to have the students focus on the operating systems concepts instead of the embedded computing aspect.

V. Assignments

In teaching the course, the assignments are incremental in building up to the final project of constructing the music player. It is important to note that although the Arduino excels in its flexibility in attaching to different shields, no additional shields are connected to the Arduino Mega until the final assignment. The goal of utilizing the Arduino Mega in this course is for its straightforward AVR CPU features. We list all the assignments here and the first 4 assignments utilize the Arduino Mega purely as a computing device.

A. Arduino I/O and Debugging

The goal of the first assignment is to construct basic functions to print out strings and numbers over the serial terminal (which is transmitted over the USB connection).

```
System Statistics

System time: 28 seconds
Number of Threads Running: 2

Thread 0
   ID: 0
   Name: blink
   PC: 0x009D
   Bottom of Stack: 0x04B0
   Top of Stack: 0x0440
   SP: 0x040B
   Stack Size: 112 bytes
   Stack Usage: 53 bytes
Thread 1
   ID: 1
   Name: print
   PC: 0x00BB
   Bottom of Stack: 0x055E
   Top of Stack: 0x04D0
   SP: 0x049B
   Stack Size: 142 bytes
   Stack Usage: 53 bytes
```

Fig. 1. Sample screenshot from the running operating system. Currently the system is running 2 threads: one to blink an LED and another to display these statistics. As the system is running, statistics update in real time.

One of the challenges with developing an OS is providing a suitable debugging environment for students. Debugging for all assignments is done over the serial connection and an aspect that is very helpful is the use of VT100 codes in communicating over the serial terminal. Many students have not used VT100 codes and they come to appreciate VT100 features such as the ability to change the text color, change the cursor position, and clear the screen. These are functions that standard VT100 codes provide and can be accomplished by sending the correct code over serial. Once the program is flashed to the Arduino and starts running, students connect using a standard terminal program.

B. Supporting Threads

This next assignment is a large conceptual leap in that it requires the students to implement a preemptive multithreading kernel on the Arduino AVR CPU. Such a large step in assignment difficulty is necessary because of the requirements of subsequent assignments. Because of the limited memory, the system is limited to 8 threads with preemption occurring every 10ms.

Using a built-in timer and its associated interrupt, threads are interrupted once their 10ms quantum expires and the interrupt code will then proceed to save CPU register state and the stack pointer. At that time, the kernel will then run the scheduler to select another thread and load in that thread's state.

Through this exercise, the students learn the intricate details of saving register state, scheduler policies, and interfacing C with assembly-level programming. Example output of what the Arduino Mega will display over the serial terminal is shown in Figure 1. This example is currently running 2 threads with the first thread blinking the on-board LED and the second thread displaying the OS statistics (over serial and updating in real time). Students are able to monitor the stack usage of each thread (at run-time) and the endpoints of the allocated thread stack space.

C. Synchronization Primitives

Once threading is supported, the next assignment involves constructing synchronization primitives. This assignment requires developing mutexes and semaphores. Threads that are waiting on mutexes and semaphores are placed into a waiting state where they are not scheduled until they are ready. In addition, students implement the capability for a thread to be set to a sleeping state.

From this assignment, students learn about the difference between busy-waiting spinlocks and having threads placed into a waiting state when waiting for a mutex. They learn that once a mutex is released, the waiting thread then can obtain the lock and continue executing.

D. Reading a Linux Filesystem

This assignment is written on a standard Linux machine and does not involve the Arduino Mega. In the subsequent assignment, it will be ported to the Arduino Mega platform.

For this assignment, the students are provided a file image formatted using the `ext2` filesystem. Modern Linux distributions are often installed using the `ext4` filesystem which contains extensions and capabilities beyond `ext2`, but for the purposes of the class, the `ext2` filesystem is sufficient.

Through this assignment, students learn the concepts of the superblock and inodes, how to traverse a filesystem to find a file, and how file data and directories are indexed using inodes. For this assignment, the filesystem image block size is formatted to match the block size of the SD card to make it easier to port code for the next music player assignment.

E. Building a Music Player

This final assignment provides the connection between programs (the threading kernel, synchronization primitives, and the filesystem reader) the students have been developing throughout the course. The music player reads 22KHz PCM, 8-bit audio files from an SD card which has been formatted using the `ext2` filesystem. As the audio files are encoded as PCM, the data can be read from each file byte-by-byte and then directly sent to a PWM pin. The duty cycle on the PWM represents the amplitude of the audio signal. The SD card itself is populated with open source songs allowing the image to be readily distributed.

Many SD card Arduino shields are readily available and the majority use the SPI protocol to communicate with the card. Because of the complexities involved with reading an SD card, a library is provided to the students that provides a clean API and abstracts away the register-level complexities of using SPI.

The challenge in building the music player is that the playback thread must be running concurrently while the data is being read from the SD card. The playback thread is scheduled to run at 22KHz (OS timer ticks occur at 22KHz) and it runs at the highest priority (it is run first every timer tick). This turns the OS effectively into a real-time OS. After the playback thread has run, the thread that handles reading the SD card is scheduled. This thread reads the SD card and fills one of two 256-byte buffers used to buffer the playback and read-ahead data in the music stream. When the buffering read is complete, any remaining CPU time is allocated to updating the user display. A functioning program has smooth playback without skipping and the program displays the current song and time.

The circuitry required to generate audio is a single capacitor and a single resistor. This basic circuit is shown in Figure 2 and is required to filter out the frequency of the PWM pin and allows just the audio frequencies to pass. They are connected to a single PWM pin which generates a digital PWM signal that becomes filtered through the low-pass filter.

A standard SD card shield is first placed on the Arduino Mega and a breadboard shield is then stacked on top of the SD card shield. Once the program is running, 2 short jumper wires can connect the ground and output terminals to a pair of headphones. Once the final connections are made to the headphones/earbuds, the 11KHz audio can be clearly heard.

Students are required to support basic music player functionality such as skipping to the next song and display of the current playing time. Because the VT100 serial display is used for program status display, the students can customize the look and feel of their music player.

VI. Experience

This set of assignments was used in Spring quarter 2018. The students taking the operating systems class are at the upper undergraduate level and consist of computer science and computer engineering majors.

One of the concerns of utilizing a very 'embedded systems' type set of assignments was the possibility that the computer engineering students would perform better than the computer science students. This was not found to be the case. The assignments themselves do not require any electrical background and the hardware component of the music player is so minimal that students of both majors were found to be capable of completing the project. The abstraction of the hardware registers seems to negate any advantage that one major would have over another in preparation.

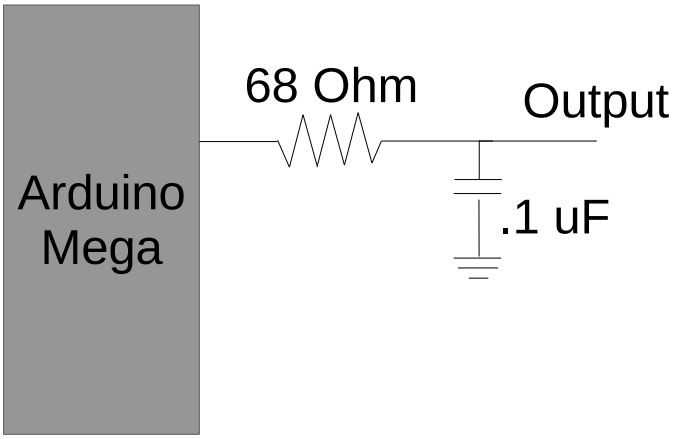

Fig. 2. Schematic of the audio circuit. It requires only 2 low-cost passive components and no additional power from the Arduino Mega. The components form a low-pass filter removing the PWM frequency.

A side benefit of the project has been the exposure of students to C and assembly in the threading assignment. In our curriculum, students do not often get experience in combining code from multiple languages into one project. Class assignment often involve a single language due to the nature of the class. For these assignments, the students were required to combine C programming with assembly level inline instructions. Prior to the course assignment, surveyed students noted that they had not mixed C and assembly in previous courses. This greatly added to their knowledge of the intricacies of how the stack works and how function arguments are passed.

The most challenging assignment has been the second assignment where the students write the multithreading kernel. This is the core of the operating system and requires a sufficient understanding of how a context switch occurs. Students find it challenging to understand the context switch process and also how to initially start a new thread. These details are implemented in assembly as well which adds to the challenge, but once these hurdles are overcome, the stack management portion is relatively straightforward.

Overall, students appreciate the ability to build a system from scratch with the final reward being a hardware music player of their own construction. Preliminary student feedback conveys the fact that they appreciate developing code for a real system instead of a virtual machine. Watching the system run and switch among threads in real-time is quite satisfying. Students were motivated by the end goal of building a system where they could listen to music. Suggested feedback has primarily revolved around the difficulty of debugging the Arduino. This has been centered around the fact that standard `printf()` function calls are not available and that has made debugging difficult. This has always been a challenge in the operating system development process and it requires the students go about debugging and testing in a methodical manner.

There are drawbacks though to developing for a system without a supervisor mode and without a full MMU. Students are not able to fully appreciate the system call mechanism to switch from user to supervisor mode. Additionally, they are not exposed to developing with full memory protection. The authors are believers in the idea that in order for students to fully appreciate the inner workings of an operating system, they need to build a working system from scratch and that the loss of those features is worth the simplicity of developing for a microcontroller. Students have often expressed this same sentiment as they have worked on their assignments.

VII. Future Work

There are a variety of features which can be added to the current set of assignments. One possibility is to add support for dynamic thread creation and closing. The current system is designed for static allocation of the threads and with further modification, threads could be dynamically created by other threads. For thread functions that are not recursive and call just a few other functions, their stack sizes can be quite small (50-150 bytes). Because of this, threads could be rapidly created and destroyed without too much memory usage.

With the number of development boards readily available on the market at the moment, a future direction could be to port the assignment to other Arduino microcontrollers (the Arduino Due and Arduino Zero) as well as other platforms that a particular institution may have already invested in. Some viable alternatives include the ST Nucleo [8] or the TI MSP [7] series of CPUs. Because of the flexibility of assignments, in terms of hardware, the microcontroller only need support for timer-based interrupts (preemptive scheduling) and a hardware PWM output pin (for audio output). For software, the development toolchain should support the ability to generate interrupt service routines that do not have a prologue or epilogue.

VIII. Conclusion

In this work, we describe a sequence of assignments for use in an operating systems course. The projects utilize an Arduino Mega and start with developing functions for serial terminal communication and end with a music player than can play 8-bit PCM encoded files from an SD card.

Through the assignments, students develop a mini-OS kernel that supports threads, synchronization primitives, and can read the `ext2` Linux filesystem.

We find that students develop skills in combining C and assembly instructions while working on the programs. The students learn an appreciation for operating systems concepts when the system is running and the thread state is visualized over the serial display. The final project of developing a music player which plays songs from an SD card is motivating for the students and upon completing the project they have learned a variety of OS related concepts.

References

[1] Arduino mega 2560 r3. https://store.arduino.cc/usa/arduino-mega-2560-rev3.

[2] Arm cortex-m series. rev://www.arm.com/products/processors/cortex-m.
[3] Baking pi - operating systems development. http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/.
[4] Chibios homepage. http://www.chibios.org.
[5] Femtoos - rtos for small mcu's like avr. http://www.femtoos.org.
[6] Freertos - the freertos kernel. http://www.freertos.org.
[7] Msp low-power microcontrollers. http://www.ti.com/lit/sg/slab034ad/slab034ad.pdf.
[8] Stm32 mcu nucleo. http://www.st.com/en/evaluation-tools/stm32-mcu-nucleo.html.
[9] M. Black. Building an operating system from scratch: A project for an introductory operating systems course. In *SIGCSE*, 2009.
[10] P. Buonocunto, A. Biondi, M. Pagani, M. Marinoni, and G. Buttazzo. Arte: Arduino real-time extension for programming mutitasking applications. In *Symposium on Applied Computing*, 2016.
[11] M. El-Abd. A review of embedded systems education in the arduino age: Lessons learned and future directions. *International Journal of Engineering Pedagogy (iJEP)*, 7:79–93, 06 2017.
[12] P. Jamieson. Arduino for teaching embedded systems. are computer scientists and engineering educators missing the boat? In *International Conference on Frontiers in Education: Computer Science and Computer Engineering*, 2010.
[13] P. Jamieson and J. Herdtner. More missing the boat - arduino, raspberry pi, and small prototyping boards and engineering education needs them. In *International Conference on Frontiers in Education: Computer Science and Computer Engineering*, 2015.
[14] P. Yakimov. An introductory embedded systems teaching using open-source hardware and software platforms. *Journal of Communication and Computer*, 13:11–18, 2016.
[15] Z. Youssfi. Making operating systems more appetizing with the raspberry pi. In *Frontiers in Education Conference*, 2017.