1 Introduction

Operating Systems are used everywhere in the modern day. Nowadays, even when performing trivial actions like ordering take-out food or checking out at a supermarket, people engage with an operating system. These systems are so ubiquitous that people outside of the field of computer science may not even realise that they are interacting with one. If you were to ask these people what an operating system was, they would probably give you an answer like, Windows [3], MacOS [2] or Linux [5]. In reality the definition of an Operating System extends further beyond the bounds of OS's for personal computers. At its core an Operating System is the low-level software which supports the basic functions of a computer, tasks such as scheduling, and controlling IO devices. Understanding the role Operating Systems play for modern computers should be important for high level programmers as the code they write will always be subject to the Operating Systems management of the computer. ARM provides a good platform for understanding Operating Systems as they allow an experience of a system before any software is present. This completely clean slate to build on can build a programmers understanding of an Operating System as before any application code can be written, the programmer has to tackle at least some of the problems.

1.1 Project Goals

At the start of the project I derived three main goals.

1.1.1 Objective 1

The first goal is to support the basic function of a computer. This includes providing an environment for user code to be run in and the tools required for a user access privileged components of the chip. This includes: Providing an Supervisor Call handler (SVC) to service calls to a graphical output; Providing a reset handler to reset relevant parts of the memory to a workable state; Providing access to input devices.

1.1.2 Objective 2

The second goal is to design, develop and interface a virtual keyboard with the system. This keyboard should at a minimum provide the ability to convey keystrokes to the system. A more sophisticated implementation would be able to recognise combinations of keystrokes and report the use of control keys being pressed to move the cursor. I propose to use a virtual keyboard as I intend the emulate an arm processor rather than develop on a real one.

1.1.3 Objective 3

The third goal is the development and implementation of a thread management system for my processor. This would differ slightly from the implementation of process management on modern operating systems as my thread management system will operate in a single memory space. Typical process management systems operate in separate virtual memory spaces. While it may be possible to implement this on an arm chip, the lack of hardware support on the particular chip I have would make this an incredibly hard task to accomplish with only software. Therefore, I opt to develop threads rather than processes. In addition to this management system, I am also required to develop the required methods to give the user access to use the management system as a tool. These methods would include the ability to create and end threads. Further implementations could include the ability to enable smooth communications between threads.

2 Motivation and Background

2.1 Motivations

The motivation for this project stems from previous courses I have taken such as COMP15111, Fundamentals of Computer Architecture, COMP22712 Microcontrollers and COMP15212 Operating Systems. When taking these course I really enjoyed the challenges behind working within an ARM based environment, such as working with few ‘variables‘ and having no pre-made software available to you. For me, these challenges raised the question of how viable it is to write an operating system for a microcontroller. While I had done a simple form of this for COMP22712, I wanted to take it further by implementing more complex features. In addition to this I wanted to improve upon the work I had done in COMP22712. This work had been relatively rushed and messy as I was having to learn on the go, and I did not have much time to re-factor. From this I derived two main goals for this project; I wanted to develop an OS which was better structured, and I wanted to develop some sort of process management service for the ARM chip. The operating systems course should help with the development of the process management service, as the notes I have explain how operating systems manage processes and threads within an OS. This project is more concerned with developing threads rather than processes, the distinction being that a thread is usually a segment of a program running as a process.

Finding ways of keeping my work organised became a large part of this project for me. Most of my programming experience has, until now, been focused on higher level languages such as C# and Java. Due to this, developing for a low level language felt quite jarring due to the following characteristics of ARM; Program structure can become disorganised and hard to read without self-imposed discipline; Braces and indentation are not enforced, which would usually expose the control flow of the program; Type checking does not exist. These problems reinforced the necessity of commenting in all my code, even beyond ARM. Consistently commenting an a specific style also became a key strategy to ensuring my code was readable. I often found myself trading off readability against efficiency.

2.2 Background

The system I built has it roots in the COMP22712 course, and it derives much of its environment from the work done there. The system was built for the graphical debugger named ‘Komodo‘ [1]. This acts as a ‘front end‘ for ‘back end‘ processor models. In my case the model I used was and emulator called ‘Jimulator‘. It provides me with the assembler for my ARM code, as well as the ability to load and run the assembled code on a virtual ARM processor. The debugging facilities it provides allow me to pause the code at breakpoints I set and inspect the state of the processors memory and registers. My working environment also consists of a few plug-ins for Jimulator provided by the COMP22712 course. These plug-ins are as follows:

- An 8 bit clock-based timer exposed as a byte in memory.
- 3 x 4 Virtual keypad with an interface set-up in the memory.
- A 320px x 240px virtual RGB display with an interface set-up in the memory of the arm chip.

These plug-ins for the Jimulator emulator expose themselves at static memory addresses with various interfaces. The timer appears as a read-only memory location at 0xF100_1010. This timer is free-running and increments at 1kHz. The keypad simulates a matrix keypad. It works by allowing the arm chip to activate scan lines and read the resulting input to determine which keys have been pressed. It also appears as a single byte containing both the scan-line bits, and the resulting output bits. The virtual screen is made of a vscreen program, connected to the Jimulator via a plug-in. The plugin exposes the access to the screen via a frame buffer set-up in memory starting at 0xAC00_0000. The memory set-up consists of 3 bytes for each pixel running left to right, top to bottom. The 3 bytes represent the RGB values of the corresponding pixel

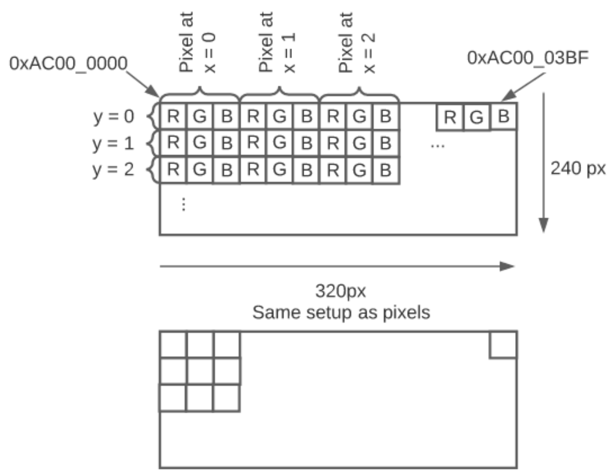

Figure 1: Frame buffer memory set-up

The system I created required me to modify this starting environment by replacing the keypad with an new plug-in, which created and handled a new virtual keyboard as detailed later in section 3.