My Little Operating System

Sam da Costa

April 14, 2022

Contents

1 Introduction

2 Motivation and Background

The motivation for this project stems from previous courses I have taken such as COMP15111, Fundamentals of Computer Architecture, COMP22712 Micro-controllers, 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 made for 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 hand 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 and OS which was easier to read through and keep organised, 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 should help, as the notes I have detail 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. Arm code is very hard to keep organised as it doesn't have visual aids such as braces to expose the control flow of the program. This can make it very hard to read and edit. In addition, you can only name memory locations. Not being able to name registers means that you have to keep track of what is where at all times. All of these problems reinforced the necessity of commenting in all my code, even beyond ARM. When I wrote the OS, I had to keep my commenting style as consistent as I could, and name variables appropriately. Very often I found myself trading off readability against efficiency

3 The Virtual Keyboard

3.1 The development environment

The virtual keyboard took far more time than I had initially anticipated. It required me to delve into the installation process for Komodo as I had no knowledge of how to attach software to Komodo. When browsing through the set-up folder and bash script for Komodo, I found the Jimulator plug-ins which my tutor had pointed me in the direction of. These plug-ins act as event handlers for the ARM chip. They can respond to things like SVC commands, memory reads and memory writes. The useful thing about these plugins is that they can also affect the chip in various ways by causing interrupt signals and writing data to specific registers. This is essentially how all the input and output was handled in COMP151111. In this course the students would make an SVC_0 call which Jimulator would intercept, and then output R0 to the built-in terminal. In much the same way a call to SVC_1 would be intercepted by Jimulator causing a read from the terminal into R0. The scripts included in the default installation of Komodo also included a virtual keypad, a timer, and a virtual screen. I had used the screen and keypad before, but I wanted more keys for the keypad, in order to make inputting letters easier. Naturally, the keypad plug-in became my reference for creating a plug-in as it was the closest in purpose for what I wanted to achieve.

I determined from the keypad plug-in that the actual interface seen by the user was a simple python script executed during Komodo's start-up. This script parsed an XML file which defined the layout of the keypad. This script then also passed the states of the buttons to a piece of shared memory. The shared memory was then attached in the plug-in script and when ever a read was made to the keypads' location, the plug-in would intercept it and update it according to the contents of the shared memory. The last bit of knowledge I needed was how to compile the plug-in, into something jimulator could actually work with. From reading the make file I determined that the code had to bed compiled with the –shared flag to compile a shared object and the `fPIC` flag to indicate it is a library and may be executed from anywhere so that its jumps need to be calculated relatively rather than absolutely.

3.2 Design

3.2.1 Virtual Keyboard Hardware Interfaces

The virtual keyboard design is very similar in concept to the virtual keypad explained above. However, it differs hugely in how it appears as a peripheral in the ARM environment. The keypad used in the COMP22712 labs appears as a single byte, in which bits 7-5 are set in turn to allow bits 3 - 0 to be scanned into. This set-up requires the system to scan they keyboard themselves and denounce the result. While this is also a possible protocol to implement with the larger keyboard I built, I felt it made more sense to have the virtual keyboard appear as 3 bytes, one to trigger the data change, one to represent an ASCII character, one to represent the direction of the button interaction (pushed or unpushed).

The advantages of handling the keyboard this way are that I don't have to have as much processing on the ARM side. This is very helpful as the code required to scan a virtual keyboard can be quite cumbersome and time-consuming. This interface also allows me to mimic modern hardware, which would send ASCII codes, rather than requiring the processor to manage the IO more manually.

3.2.2 Keyboard Design

I used the python library glade to model the keyboard. Unfortunately, lots of these buttons are there for appearance's sake only, as this keyboard does not implement a full ASCII character set. I chose to do this as I felt the time to benefit ratio did not warrant spending more time on this. While I could have implemented more control characters, I would be highly unlikely to use them so, I felt it would be wasted time.

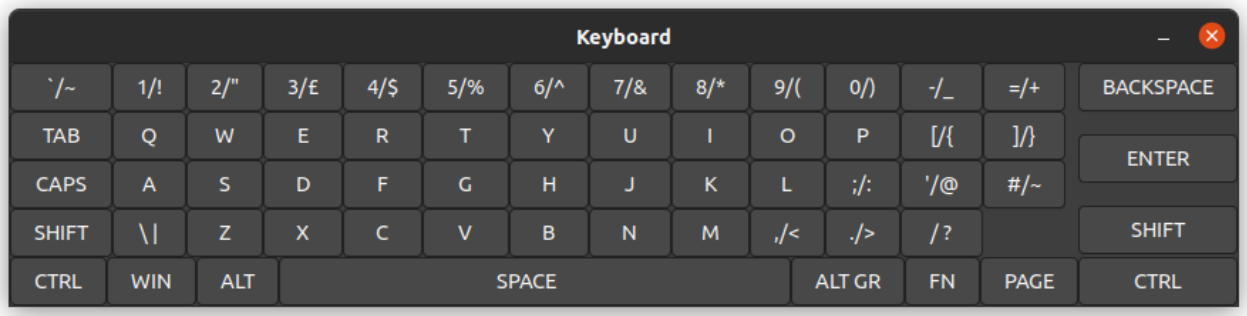

Figure 1: The final design of the virtual keyboard.

I managed to implement working characters from ASCII 0x20 to 0x7F as well as working caps-lock, shift, backspace, enter and tab characters.

3.3 Implementation

The initial protocol I developed to handle input is as follows.

- The user presses a button on the keyboard
- Glade signals the plug-in that a key has been pressed (or unpressed) via the shared memory
- The plug-in reads the data and make it ready to write.
- The plug-in throws an interrupt.
- The interrupt routine receives the interrupt and writes a 1 to the request register
- The plug-in writes the ASCII character and a 1 or 0 depending on whether the button was a 'push' or 'unpush'
- The IRQ routine writes the data to a map of keys
- SVC handler can now be called to read pushed keys from the keyboard map

This protocol changed in the final version, as I incorporated the ability to suspend the current thread to wait on IO.

4 The Basic OS

4.1 Layout and structuring

As mentioned in Chapter 2, one of the main problems I had in COMP22712 was keeping my code organised as I was learning the intricacies of ARM while trying to write code. Now armed with a little more experience, I wanted to ensure that my code was kept organised from the start. I decided the best way to start would be to organise my main file `os.s`. This file in my work for COMP22712 had been quite disorganised. It had a lot of the handlers required in the same file, which I felt was not great practise as it does not allow for a separation of concerns. I organised this file by making use of the INCLUDE mnemonic. This mnemonic is similar in concept to an import command in Java or Python except it differs in its exact implementation. INCLUDE in ARM has the effect of moving the code from the specified file to the location of the INCLUDE command. This is close to how C implements include. This separation of concern leaves the file `os.s` as a file which starts as a vector table, containing branches to each of the handlers, followed by a long list of include statements defining where the handlers are kept and where prerequisite files are kept. The file also includes a halt loop which I can branch to if something happens which I cannot recover from. This really helped with debugging as I could use this loop to stop the processor after an error without it changing the state of any memory addresses. An issue I had encountered a lot in past course was that when I would run into something like a data abort or an undefined instruction, I would have nothing to stop the processor from overrunning the handler it would jump to. This would quite often make things hard to debug. Having the halt instruction allowed me to give the processor some control over halting itself.

4.2 Self Imposed Conventions

In order to keep my code organised and readable, I picked up a few conventions along the way which I tried to stick to. These were chosen with the intention of making my code easier to update in the long run as ARM is a difficult language to read. One of the conventions I stuck to best was to comment every procedure call under the label with a definition of which registers are used for input and output. This format provided me with an easy way to look up my method's and determine how to use them. Another benefit of this format is it distinguishes the branch label from other labels as an actual procedure call. Another convention I employed was the consistent pushing style of registers. When a procedure starts I always immediately push the LR, regardless of whether I need to. This is so that if I require a call to another procedure call, I don't need to remember to push the LR as it is already done. If I hadn't done this, then each time I needed to add a nested procedure call, if I forgot to push the LR then I would have an error on my hands which I would likely have to hand trace to debug. Similarly, when writing a new procedure, I would also push the registers I need to work in, and then immediately pop them, and then write the procedure in between. This meant that I could more closely mimic writing in a higher level language as I didn't have to think as much about the unusual parts of ARM. An example of these conventions is described below

Listing 1: How all of the procedure calls started

```
queue_index
; IN  R0 - index to check
; IN  R1 - Pointer to queue
; OUT R2 - item to return or -1 if invalid
PUSH {LR}
PUSH {R0 - R1}
PUSH {R3 - R12}

; Actual procedure code goes here.

POP {R3 - R12}
POP {R0 - R1}
POP {LR}
MOV PC, LR
```

I also utilised the `EQU` directive often, in order to aid the readability of my code. For any constant or immediate value used (other than simple values such as -1, 0, 1, 2, 4) I would aim to name label them, and then only use the label. Another benefit of using this directive is that I could use them to do arithmetic to define how much space I would statically assign to blocks of memory. This was useful when designing the process control block, as I could scale how much memory I would need according to a single constant MAX_THREADS. The ability to perform arithmetic operations with aliased names made scaling the program much easier. Finally, the last convention I imposed on myself was to write a commentary along some of the more technically challenging aspects of the code. For example the context switching procedure is the most complex thing I've written in ARM if not in all languages. So while writing this code I would start by writing a short comment describing the small subtask I wanted to complete, before actual writing the code to complete this subtask. This was quite a time-consuming process, as any changes which I needed to make usually meant that I had to re-write my comments. Due to this, I only employed this strategy when it was really necessary. I found this so helpful, that I would often describe the problems I needed to overcome for a specific subroutine in the subroutines' header. This acted as a cheaper way of documenting my code well without spending too much time on it.

4.3 Virtual LCD

The installation of Komodo which I was developing for had a virtual LCD which I could manipulate from ARM. I wanted to provide some methods to the user which could be used to manipulate the screen. The screen appears in memory as a large frame buffer. This is in contrast to how the real world alternative does, as the real world alternative provides methods in which to output ASCII characters. As the virtual LCD does not contain these methods, I have to make them myself. I have done this before in COMP22712, however I felt I had an opportunity to improve upon this code. The past code I had written had some issues. For example the print character function did not actually work as intended. It had no ability to print control characters, which I felt was an oversight. My previous code also could only print in black and white which I felt was somewhere I could improve in. I decided that the best way to approach this task was with a top-down approach. Accordingly, I set about developing the print string function, which contained a call to print char, which I would develop later. This procedure was quite simple, as all it needed to do was loop through a string and call print char on each character, until it reached the `NUL` character. Then I could develop the print char procedure. This procedure followed the following protocol listed in Listing 2

Listing 2: printchar psuedocode

```
if ( char is a control character ) {
        update cursorposx according to char
        correct to ensure 0 <= cursorposx <= 40
        update cursorposy according to char
        correct to ensure 0 <= cursorposy <= 30
} else if ( char is a letter ) {
        print the character
        update cursorposx
        update cursorposy
} else {
        halt the processor
}
```

The challenge of writing characters to the LCD essentially boils down to two main problems - Outputting a character template and keeping track of where the cursor is. Both problems are relatively trivial to solve, however how to write code to solve them efficiently is difficult.

4.3.1 Keeping the cursor position consistent

I solved this problem by first handling the control characters. I chose to determine which control character I was working with via a simple jump table. This has the benefit of allowing me to add more control characters very easily. Once I have jumped to the correct position I can then perform the correct operation. From here I then update the cursor position by performing the update and then checking and correcting the x and y coordinate against the bounds of the screen. As similar method is employed to correct the cursor after writing a character. Essentially every operation on the screen should leave the cursor in a position ready to print a new character. The characters I have supported are listed, and their effect are seen in Table 1.

Table 1: Supported control characters.

Backspace	Delete a character left of the cursor
Horizontal Tab	Move the cursor right
Line Feed	Move the cursor down one line
Vertical Tab	Move the cursor up one line
Form feed	Clear the screen
Carriage return	Move the cursor to the start of the next line

4.3.2 Outputting a character

To output a character I use a 7 x 8 bit font which was provided in COMP22712. It provides a font for ASCII characters 32 to 126. To determine the address of the font I have to subtract 32 from the character to normalise the character to the base of my font map. I then multiply by 7 bytes to determine the correct address. From here I have to read the loop over the 7 bytes as a 2d array essentially, with one dimension as the bytes and one dimension as the bits.

4.4 The SVC Handler

My SVC handler was one of the few pieces of code which I felt I could salvage from my work in COMP22712. The handler provides an organised way to interpret an SVC instruction and

determine which operation to direct the processor to. It determines which program to jump to by reading the instruction in the link register. The handler then clears the opcode, leaving a 24 bit value which represents which program the SVC commands is referencing. It checks this code against the SVC_MAX constant. This is a security measure to ensure that the SVC command cannot branch to any arbitrary code. In a single ADD instruction it then multiplies the SVC constant by 4 to get a words address and then adds it to the PC. The next instruction loads the address at this address to the program counter which causes the handler to jump to the correct position. This jump table method is an alternative to the most simple form of SVC handler which is a long string of 'switch style statements'. I chose the jump table over this method as I needed to support 12 methods, so a long chain of switch statement would not be particularly efficient. The operations I supported are as follows:

halt	(Halts the processor)
printchar	(Prints a character to the virtual LCD)
printstring	(Prints a NUL terminated string to the virtual LCD)
timer	(Copies the timer into R0)
button data	(Gets the data from the virtual buttons *deprecated*)
setcursorposx	(Sets the horizontal position of the cursor)
setcursorposy	(Sets the vertical position of the cursor)
query_keyboard	(Grabs the first pushed key from the virtual keyboard)
query_key	(Checks if a specific key is pushed)
create_thread	(Starts a thread from a specified address)
end_thread	(Kills the current thread)
halt_thread_for_IO	(Halts the thread until input occurs and then runs query_keyboard)

My SVC handler also includes a brief exit procedure which is always jumped to after completing an operation. This procedure just re-enables interrupts as I disable them during the SVC entry procedure to ensure that the operations execute atomically.

4.5 The IRQ Handler

5 Implementing Thread Switching

5.1 The challenges of thread switching

That challenges of thread switching in ARM stem from the difficulty in storing a threads' context in a way which is recoverable from any point without losing any state. A good place to tackle this problem is to start from looking at how a procedure call works in ARM. Procedure calls in arm are somewhat similar in concept, but they differ in scope. The procedure calls [1] I used in ARM consisted of the following steps

- Move the LR onto the stack
- Push non-parameter registers
- Execute the required procedure
- Update the defined output registers
- Recover the non-parameter registers from the stack
- Pop the LR back
- Return to the call location

This structure for a procedure call allows me to nest procedure calls within each-other where necessary. Where the thread switching protocols differ is that they can't merely push registers to the stack. This is because, for each thread running, neither thread should have to concerned with the others' existence. Each thread should be able to access the resources available without any concern that the other threads could be modifying the contents of the stack or a threads registers. In a sense threads should be invisible to each other. This raises several issues:

- When should I run the context saving procedure to perform the time slicing?
- Which modes can you actually interrupt, and which modes need to be handled atomically?
- How do you keep the user's stack consistent for each thread?
- How do you store and load a threads' context without corrupting the registers or CPSR?

5.1.1 When is the time-slicing procedure called?

This is the easiest of the challenges above to solve. One of the plug-ins provided in the installation for Komodo is a configurable timer which can cause an interrupt. This is a natural entry point for the time slicing procedure, as a successfully implemented interrupt should be handled invisibly relative to the currently executing program. This is essentially what I want to occur. The executing program halts for the interrupt, which then gives control to another program, until another time slice is complete and control is returned to the original program. Of course in this example I am using two threads, but it can generalise to more threads.

5.1.2 When can I safely interrupt?

This is the next easiest challenge to solve. While it is possible to implement complex behaviours in ARM such as nested interrupts without extra hardware support, the software complexities required to implement it are usually not worth the benefits. Allowing nested interrupts can make it very convoluted (but not impossible) to return the processor state to its precise state before any interrupts had occurred. The only upside of this added complexity that I can see is that you can reduce the latency in which you handle interrupts, which I could see being useful in some situations (such as a real-time system) but not in this one. Therefore, the solution as I see it is to leave interrupts as disabled during a standard interrupt call. Similarly, I would disable interrupts during a supervisor call, as it would complicate saving the state of the processor as I would need to ensure that the supervisor stack was kept consistent. Again, this is possible to do, but it does not yield many benefits for me.

5.1.3 How do I ensure the user stacks consistency?

The solution to this problem looks more simple than it is to implement. I chose to assign each thread its own user stack to ensure that each program can operate on its own stack independently. The memory space assigned to each thread is statically assigned on start up according to a constant MAX_THREADS. This constant defines the maximum number of threads which I allow, and allocates memory accordingly and divides it between the threads. Once I have created this memory space, when I give a call to create a new thread, I can pick the first free stack space and calculate a stack pointer for it. The reset procedure also has to account for this set-up as the main thread has to be treated in same way as any other thread. This means that on reset, the correct data must be inserted into my process control block (PCB) to mimic a call to my thread creation procedure. Now that each thread has its own stack pointer, saving my stack during a context switch is as simple as saving my stack pointer.

5.1.4 How do you store and load a threads' context?

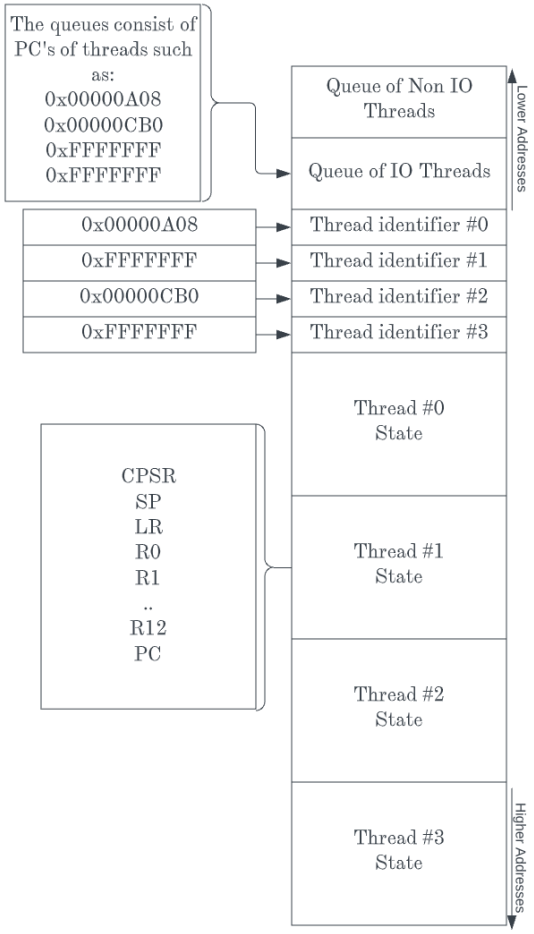

Figure 2: The memory layout of the PCB.

This is the hardest problem I encountered when implementing threading. The loading and storing of a threads' context requires me have an organised way of storing and recovering each context. This is when the PCB comes in. The PCB consists of the setup as seen in Figure 2. The PCB represent the state of saved threads at any given point. A threads state is represented by first storing its PC in either the IO thread queue or the Non IO thread queue, depending on whether it was created with access to the virtual keyboard. Once a thread is pushed onto a queue, its PC is used as an identifier in the thread identifier block. This block is an unsorted block, in which a thread is inserted at the first free location as designated by a -1 (0xFFFFFFFF). The index of the threads PC then acts as an index for the thread states. The thread state at that index will then hold the actual registers for the thread. The registers are stored in the order shown on the left with the CPSR, SP and LR being stored before R0 - R12 and then the PC. An option I had to consider when creating this memory structure was how I was going to identify my threads. Most modern computers use a unique process identifier (PID) to reference a thread or process. I briefly considered implementing this rather than identifying threads via their program counter however I felt that PIDs are more useful for a processor where a process might be executed on an arbitrary core rather than a single core. In my system, as there is only one 'core' nothing can change the PC without running the code, so it acts as a decent primary key.

The order of the registers is very specific in order to aid the context switching procedure, specifically the loading of the previous state. To perform the load, first the scheduler has to determine which thread to revive, and then get a pointer to the CPSR in the thread state. Once it has this pointer and has taken the thread identifier off of the queue and the thread identifier block it can perform the load in four simple instructions.

Listing 3: Return Procedure.

```
LDMIA R3!, {R4}
MSR SPSR_c, R4
LDMIA R3!, {SP, LR}^
LDMIA R3, {R0 - R12, PC}^
```

These instructions work by manipulating the address in R3 which points to the CPSR. The first instruction loads the CPSR into R4, and writes back the increased address to R3. The second instruction updates the SPSR, so that when the mode is switched the threads CPSR gets updated

correctly. The third instruction copies the SP and LR into the user's SP and LR and the writes back the incremented address. The final instruction loads the threads registers including the PC and causes the SPSR to be copied into the CPSR.

The procedure to store threads is somewhat more convoluted than the loading procedure. Once the address of the threads state is calculated, the first task is store the threads CPSR. This is done by storing the current SPSR, as this procedure is called from IRQ mode so the SPSR holds a copy of the last threads CPSR. Then to store the SP and LR, I use the `STMIA` commands with the hat. This allows me to access the user mode registers. I then have to pop the registers R0 - R12 and store them to the thread state. I then have to reset the SP to before the register R0 - R12 are pushed. This is in order to ensure that the SP is correct for the next time that IRQ mode is entered. If this step is not taken, then every time the time slicing operation is called, the IRQ stack will grow by 13 bytes. This will then cause the stack to overrun, which is unrecoverable, at least in this system.

5.2 Integration with virtual IO

6 Conclusions

7 References

References

[1] *ARM Assembly Language, 2nd Edition.* CRC Press, 2014.