My Little Operating System

Sam da Costa

April 24, 2022

Project Supervisor: Jim Garside

Contents

1 Introduction

Operating Systems [1] 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 [2], MacOS [3] or Linux [4]. 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 System's play for modern computers should be important for high level programmers as the code they write will always be subject to the Operating System's management of the computer. ARM provides a good platform for understanding Operating Systems as it allows 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 was 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 system. This includes: providing an Supervisor Call Handler (SVC) to service calls such as calls to graphical output, providing a reset handler to reset relevant parts of the memory to a workable state and providing access to input and output devices.

1.1.2 Objective 2

The second goal was 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 was the development and implementation of a thread management system for my processor. This differs slightly from the implementation of process management [5] 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 a hard task to accomplish with only software. Therefore, I opt to develop threads rather than processes. In addition to this management system, I also need to develop the required methods to give the user access to use the management system as a tool. These methods 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 ‘Fundamentals of Computer Architecture‘, ‘Microcontrollers‘ and ‘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 my Microcontrollers course, 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. 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; Automated 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 Microcontroller’s course, and it derives much of its environment from the work done there. The system was built for the graphical debugger named ‘Komodo‘ [6]. 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. 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 processor 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.

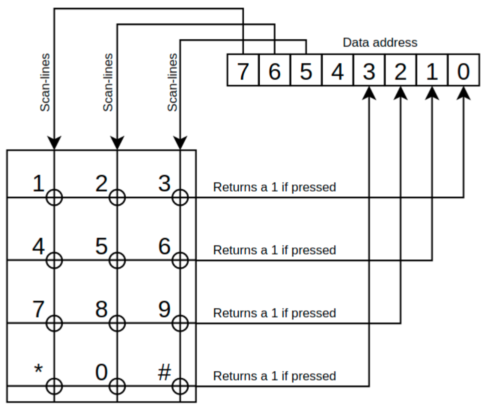

Figure 1: Keypad Matrix Setup

The virtual screen is made of a vscreen program, connected to the Jimulator via a plug-in. The plug-in 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

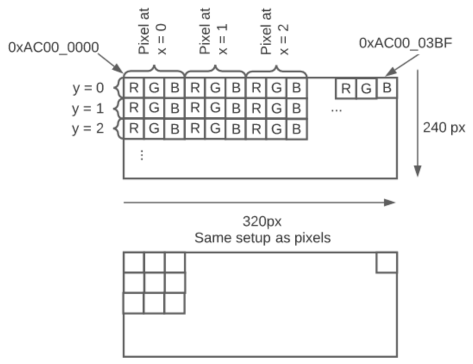

Figure 2: Frame buffer memory set-up

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.

3 The Virtual Keyboard

As mentioned in section 1.1.2, one of the goals of this project is to provide the system with a way of accepting keyboard input from the user. This is in service of creating peripherals for the operating system to manage. It also provides a convenient way of demonstrating the thread management system as in real operating systems threads will often suspend themselves waiting for input.

3.1 UI design

The keyboard was based of my real world keyboard. Due to this many of the keys are more decorative than practical as they do not all have an ASCII [7] equivalent. The layout was designed in glade [8] and saved as a XML file which could be loaded by a python script. Glade was a helpful tool to perform some of the less technical work for this section. The glade editor provided an easy way of laying out the keyboard on a grid, a specifying an ID for each button.

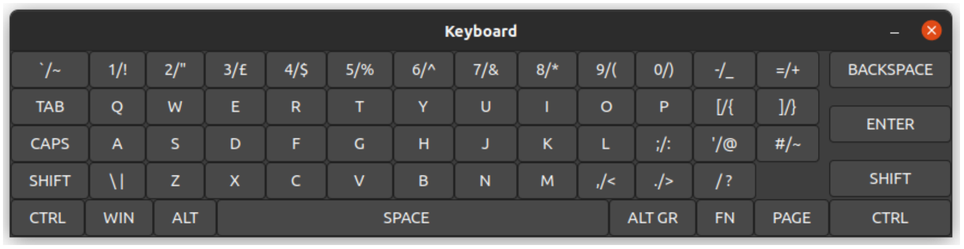
Keyboard
`/~ 1/! 2/" 3/£ 4/$ 5/% 6/^ 7/& 8/* 9/(0/) -/_ =/+ BACKSPACE
TAB Q W E R T Y U I O P [/{]/} ENTER
CAPS A S D F G H J K L ;/: '/@ #/~
SHIFT \| Z X C V B N M ,/< ./> /? SHIFT
CTRL WIN ALT SPACE ALT GR FN PAGE CTRL

Figure 3: The final design of the virtual keyboard.

3.2 Implementation

3.2.1 Python Script

The virtual keyboard is implemented as a python [9] script which runs in a separate process forked by Jimulator at runtime. This script simply loads the XML file produced by glade and runs the window seen by the user. It then continuously loops, checking for any change in state of the buttons. It can then use this data to report any state changes. These data changes are reported to Komodo via shared memory, set-up in this script.

3.2.2 Jimulator Plug-in Integration

The Jimulator Plug-in is called by on Komodo loading. This C++ plug-in [10] forks and executes the python script. It then attaches the shared memory in the python script to allow for communication between the glade application and Komodo. This plug-in then creates the interface between the ARM code running and the data generated by the glade keyboard. The interface is formed of 3 main registers assigned statically in the ARM chips memory. The first two registers are for transmitting the ASCII key pushed and the direction of the push (Up or Down). The third register is written two by the Operating System to acknowledge the key-press and free up the registers for new data. On a key push, plug-in will throw an interrupt and load the key data and key direction data into the correct registers. The Operating System will respond to the interrupt by reading the data, passing it to an application and clearing the interrupt via the interfaces acknowledge register. The specifics of passing the data to the application depend on the application method of querying the keyboard. There are two methods, ‘polling‘ and ‘interrupt based‘. For both of these methods the plug-in works in the exact same way, the only difference is how to application code reads the data. When an interrupt is thrown and handled, by the keyboard, the data is written to a map in the Operating System’s memory. The application can then either choose polling, in which the application continuously reads this map until a key-press is found, or it can choose the interrupt based method, in which the thread running the application suspends itself pending the keyboards next interrupt. When this interrupt is seen the application code reads the map which will have a key press in it and then returns to execution. This process is described in depth in section 5.2

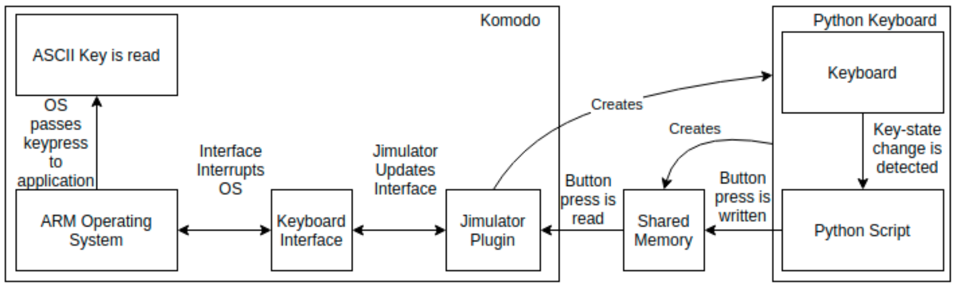

Figure 4: The full integration of the virtual keyboard with Komodo.

3.3 Evaluation and Improvements

The keyboard does accomplish Objective 2 to some degree of success. It does reliably dispatch keys to the Operating system to handle. However, there are some minor improvement which I would have liked to have made. For example, this keyboard contains a single buffered item. Most modern keyboards have a larger buffer so that it is possible to queue up keys in the buffer. This ensures that the Operating system does not miss keys due to a full buffer. My current implementation will grab the first key and then not be able to read more items due to the full buffer. This makes the prompt reading of the buffer critical as keystrokes are often followed by the key being released. Clearly this is a rather significant flaw, however it does not detract from my original goal for the keyboard which was to give the operating system some IO to manage, and some reasons to suspend threads. The keyboard does actually function relatively well as a keyboard despite the small buffer due to the fact that its is a virtual keyboard only accessible by clicking. This means its quite hard to 'type' fast, so in reality the buffer issue doesn't impact its effectiveness as much as you might think. The keyboard is also able to recognise capital letters via the use of the caps-lock button as well as grammar characters and what I refer to as control characters. The control characters are the keys I used to implement the ASCII characters backspace, horizontal tabulate, line feed, and carriage return. This was one of the more difficult features to implement as it these characters require special handling. A further improvement I could make to the keep board is the ability to recognise and report combined key presses like `ctrl-A`. This would require a large rework of how glade reports the key presses, but it would not be impossible.

4 The Basic OS

4.1 Layout and structuring

As mentioned in Chapter 2, one of the main problems I had with ARM programming 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`. It had a lot of the handlers required in the same file, which I felt was not great practice 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. The INCLUDE directive provided by my assembler 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.

Listing 1: My main os.s file

```
ORIGIN &00000000
B hard_reset                          ; +0   (00)
B undefined_instruction_handler       ; +4   (04)
B svc_handler                         ; +8   (08)
B prefetch_abort_handler              ; +12  (0C)
B data_abort_handler                  ; +16  (10)
NOP                                   ; +20  (14)
B IRQ_handler                         ; +24  (18)
B FIQ_handler                         ; +28  (1C)

halt ; should be jumped to, to stop the
   proccessor
MOV R0, R0
B halt

; Import handlers
INCLUDE handlers/reset_handler.s
INCLUDE handlers/instruction_handler.s
        ...

; Import definitions
INCLUDE general/printchar.s
INCLUDE general/printstring.s
        ...
```

```
ALIGN
INCLUDE general/usercode.s
```

My `os.s` file also included a halt loop which I could jump to as a way of ‘halting‘ the processor as there is no way for the processor to halt itself 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

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 2: 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 {R0 - R1, R3 - R12, LR}

; Actual procedure code goes here.

PUSH {R0 - R1, R3 - R12, LR}^  ;return
```

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 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 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 Screen

The installation of Komodo which I was developing for had a virtual screen (as described in figure 2) which I could manipulate. 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 most real world alternatives would present themselves. For example, if you were to use a Hitachi HD44780 [11], you would find that it presents its self as a more 'intelligent' controllable device. Rather than writing directly to the frame-buffer to write characters, it is far more efficient to give it commands to write the characters, which it can then carry out itself. As the virtual screen does not contain these methods, I have to make them myself. I have done this before, however I felt I had an opportunity to improve upon this code. My previous code also could only print in black and white which I felt was somewhere I could improve in. My new functions would take a pointer to a string as a parameter as well as a pointer to 6 bytes defining the RGB colour of the background and text colour.

The procedure I developed to print a single character is shown in listing 3. This procedure would be called for every character in a string in order to provide the print string function.

Listing 3: printchar pseudocode

```
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 screen 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. These characters are the ASCII characters used to control the movement of the cursor. 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 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. A 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 is 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 pixel font. 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 as demonstrated in figure 5.

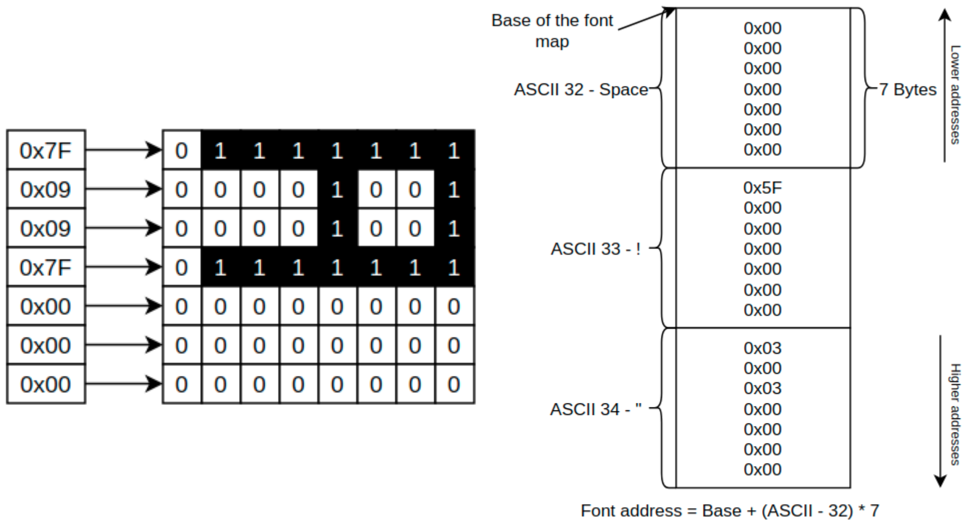

Figure 5: The font set-up in memory

4.4 The SVC Handler

The SVC handler is a program used to allow users to safely execute code which may require higher privileges. For example, if a user wanted to read data from a peripheral, this may be acceptable to the OS, but it has to be done in a controlled way. The OS provides the SVC handler as a way of enabling this access without letting the user handle the specifics. 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 will read the instruction pointed to by the LR and strip the instruction of its op code. This has the effect of leaving the parameter of the SVC instruction which can be used to compare against the implemented SVC routines. It checks this code against the SVC_MAX constant. This constant defines the highest identifier supported by the chip. 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 also described in listing 4. I chose the jump table method as I needed to support 12 methods, so a long chain of switch statements to determine the SVC parameter would not be particularly efficient. The jump table method is a far more effective way of directing execution to the correct procedure

Listing 4: The SVC handler

```
LDR R14, [LR, #-4]         ; Read the caller svc
   command into R14
BIC R14, R14, #&FF000000   ; Clear the opcode

LDR R3, SVC_MAX            ; Check user is not
   trying to execute arbitrary code
CMP R14, R3
BHI SVC_unknown

SUB R14, R14, #&100        ; Normalise base of
   SVCs
ADD R14, PC, R14, LSL #2   ; Calculate SVC jump
   point in the table
LDR PC, [R14]              ; Perform Jump

DEFW SVC_0  ; halt
DEFW SVC_1  ; printchar
DEFW SVC_2  ; printstr
...
```

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 the processor disables them during the SVC entry procedure to ensure that the operations execute atomically.

4.5 The IRQ Handler

5 Implementing Thread Switching

5.1 Thread switching in Unix

Thread switching is the process of interrupting the execution of the CPU to store its current work and load other work. When this work is in the same virtual memory space, this is known as a thread rather than a process. This mechanic can allow Operating Systems to provide more advanced features such as perceived multitasking and interrupt based input handling. In Unix, context switching is performed by saving the current process to the Process Control Block (PCB) before loading the new process from the PCB. The PCB contains all of the data relevant to each processes including the registers, stack pointers, program counter and page tables. This switching can have a significant impact on performance as caches relevant to the previous thread must be flushed as they all reference a previous thread so cache hits reduce dramatically. Allowing this context switching means that each thread can operate within a time-slice dictated by the scheduler. Each thread will be given a certain amount of time to get is work done, before it is forced to wait for other processes.

The context switch is only half of the problem with thread switching. The other half is choosing which thread to run after a switch. This problem can be solved in many different ways, and each solution is better suited for different needs. These solutions are known as policies. Policies which exist include:

- Round Robin (RR) This policy will run each thread in turn for a specific time-slice. It is one of the fairest easiest algorithms
- First In First Out (FIFO) This policy will run each thread to completion before moving to a new thread.
- Shortest Job First (SJF) This policy will run the shortest job to completion before moving on to the next shortest job. This results in lower average waiting times, but can also result in time starvation for longer jobs. This is where the longer jobs don't get any time to execute as there are always shorter job available.

5.2 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. Each thread must save its own registers, stacks, program counter, processor flags and link register. 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 [12] 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, no 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?
- When can I safely interrupt?
- 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?

5.2.1 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.

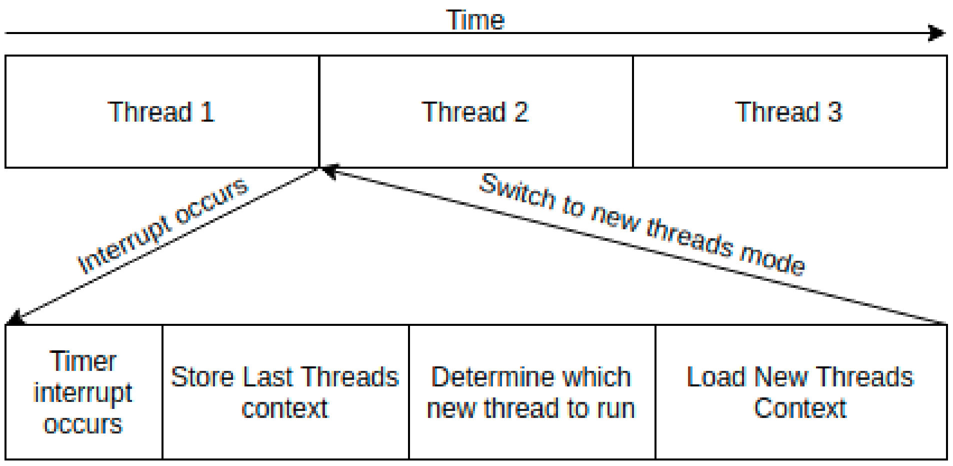

Figure 6: Context switching procedure for my system

5.2.2 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. To implement nested interrupts properly, additional hardware is needed. 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 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 is to leave interrupts as disabled during a standard interrupt call. Similarly, I would disable interrupts during a supervisor call, as it would simplify saving the state of the processor as I don't need to ensure the consistency of the supervisor stack.

5.2.3 How do I ensure the user stacks consistency?

The solution to this problem looks more simple than it is to implement. During thread creation, I 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 amongst 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 in my process control block 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 to mimic a call to my thread creation procedure. Now that each thread has its own stack pointer, saving a thread's stack during a context switch is as simple as saving my stack pointer.

5.2.4 How do you store and load a thread's context?

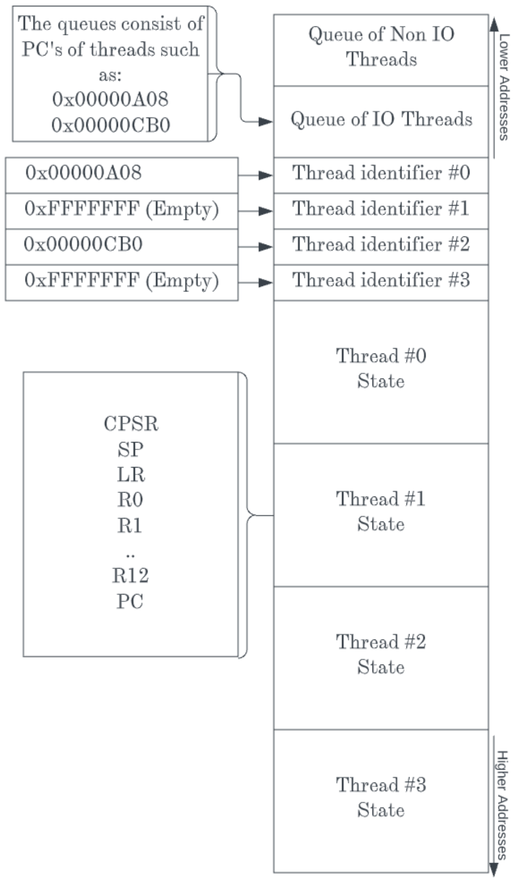

Figure 7: The memory layout of the PCB.

This is the hardest problem I encountered when implementing threading. The loading and storing of a thread's context requires me have an organised way of storing and recovering each processor state. This is when the PCB comes in. This stores all the essential data neccesseary to restore a threads state. The PCB consists of the setup as seen in Figure 7. The PCB holds the state of saved threads at any given point. A thread's 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 array, in which a thread is inserted at the first free location as designated by a -1 (0xFFFFFFFF). The index of the thread's PC then acts as an index for the thread states. The thread state at that index will then hold the 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. This system does come with some caveats which I address in the evaluation in section 5.4.

The order of the registers is specific 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 5: Return Procedure.

```
;R3 Points to the start of the thread state to be
    loaded.
LDMIA R3!, {R4}
MSR SPSR, 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 thread's CPSR gets updated correctly. The third instruction copies the SP and LR into the user's SP and LR and then writes back the incremented address. The final instruction loads the thread's 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 thread's 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 thread's CPSR. Then to store the SP and LR, I use the `STMIA` commands with the caret. This allows me to access the user mode registers. This is important as different modes have their own copies of the SP and LR which the caret enables me to access. I enter this procedure from my IRQ handler, which will push my user register to the stack as its first action. I have to retrieve these registers first by popping them. Once popped I then need to store them to the PCB via pushing. I then have to reset the SP to before the register R0 - R12 are pushed. This is 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 thirteen bytes. This will then cause the stack to overrun, which is unrecoverable, at least in this system.

5.3 Integration with the virtual keyboard

A part of the reason to implement threading was to enable efficient querying of the keyboard via interrupts enabling halted IO threads. Due to this it was important to ensure that the virtual keyboard could seamlessly integrate with the threading system. On account of this I developed a second method to query my keyboard as an alternative to my polling method. This method is composed of the following steps:

- The thread makes a call to SVC_11 to halt the thread
- The system saves the halted thread
- The system switches to another thread
- The keyboard throws an interrupt on input
- The system context switches back to initial halted thread
- The keyboard is queried.

This method uses the same method to query the keyboard as my polling method, but it differs in that it only needs to check the keyboard once as it will wait until the keyboard alerts it to new data.

5.4 Evaluation of threading system

6 Conclusions

7 References

References

[1] W Stallings. *Operating Systems.* 2011.

[2] Windows [Operating System]. `https://www.microsoft.com/en-us/windows?r=1`. Accessed: April 24, 2022.

[3] MacOS [Operating System]. `https://www.apple.com/uk/macos/`. Accessed: April 24, 2022.

[4] Linus Torvalds. Linux [Operating System]. `https://www.github.com/torvalds/linux/releases/tag/v4.1-rc8`. Accessed: April 24, 2022.

[5] Daniel P.Bovet & Marco Cesati. *Understanding the Linux Kernel.* 2005.

[6] KMD - Komodo Manchester Debugger. `https://studentnet.cs.manchester.ac.uk/resources/software/komodo/`. Accessed: April 24, 2022.

[7] The American National Standards Institute. The ASCII character set. `https://www.asciitable.com/`. Accessed: April 24, 2022.

[8] Glade - A User Interface Designer. `https://glade.gnome.org/`. Accessed: April 24, 2022.

[9] Python. `https://www.python.org/`. Accessed: April 24, 2022.

[10] Brian W Kernighan and Dennis M Ritchie. *The C programming language.* 2006.

[11] Ltd Hitachi. *HD44780U (LCD-II) (Dot Matrix Liquid Crystal Display Controller/Driver)*, 1998.

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