The University Of Manchester

B.Eng (Hons) Computer Systems
Engineering

My Little Operating System

**Jonathan David Quespaz
Sánchez**

Supervised by James Garside

Third Year Project Report

April 2019

Abstract

An operating system is an abstraction layer between the user applications and the hardware where the applications execute. It provides a simplified interface to the available resources; to achieve this several techniques are used.

The following report describes the process of designing, testing and implementing a primitive instance of an operating system, experimenting with different approaches during the development. Its contents include motivations to choose the project, assumptions made before and during the implementation, details about the hardware on which it is going to be placed are briefly discussed as well, evaluation of the results obtained from the chosen procedure and conclusions drawn throughout the development time.

Acknowledgements

I would like to express my gratitude to Piedad Sánchez, my mum; Becker Sanguano, my dad; María Ruth, Ana Belén, Sarahí Isabel, Liz Abigail, my sisters; and God for helping me and motivating me every step of the way.

Thanks to Jim Garside for being a dedicated supervisor and personal tutor, and thanks to all the staff, students and friends that have supported me during these three great years at the University of Manchester.

Also thanks to The National Secretariat of Higher Education, Science and Technology (SENESCYT), institution which made it possible for me to study in this university by the program of "Universities of Excellence" providing the whole funding of my undergraduate studies.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivations

There was a wide range of interesting options to choose a project from, and after reading their details, a few were taken aside for further consideration. Among those, topics like algorithms, security, learning tools and machine learning were investigated but the one that drew attention was the possibility to develop an operating system, not only to reinforce the knowledge gathered in previous years but to apply it in a new, challenging and practical form.

The experience gained from this project was considered invaluable as custom hardware and software are being developed every day and usually function better than off-the-shelf devices. There are advantages and disadvantages when designing and producing a tailor-made piece of software; the most important advantages are better performance, flexibility, and complete control over its features. However, in terms of resources, developing software is expensive; it takes a lot of time and the monetary cost is considerably higher compared with a tested and ready-to-run product which could be purchased. There is always a trade-off, and the decision relies on the purpose of the software to be used or developed.

However, some other options also provided the experience which is useful in a future industrial environment. Another reason to choose this project was learning to get a deep understanding on the inner workings of an operating system, how the circuitry is able to perform what, for the user, is perceived as difficult operations by just giving it some parameters, and how to map a function to do it from a high level of abstraction to the "bare metal" we have available which may consist of multiple and different devices. This process presents problems and there can be different approaches to tackle them, focusing on providing an interface to connect the needs of the application with the resources at hand.

1.2 Insight

The definition of an operating system is a matter of perspective as it is difficult to establish a boundary to define which features are included and which are not within its domain, but there is certain functionality that must be present to consider a piece of code an instance of it. [Latham and Garside, 2019]

An operating system is a layer of abstraction between the physical hardware available and the user developed software which manages resources and makes them ready to be shared among multiple programs through an intuitive, simple and consistent interface. Some of the resources which are managed by this layer are time, memory, input and output devices and others which are translated into multiple functions such as scheduling, process interaction, interrupt handling and file system management among others. [Garside, 2018]

Moreover, there are different organisational structures for an operating system to be considered and are shown in figure 1. A monolithic design approach is hard to understand, modify and maintain since everything between user applications and hardware interaction is packed into a single entity, a single block, that is why its called monolithic, but the cost of those interactions is low.

Layering a system is another approach, even though this allows the developer to test and verify each layer individually and then together which is easier than to do it all at once, there is an associated overhead needed to pass and transform requests between layers which affects performance.

To minimise the contents of the operating system the microkernel model is used, by moving part of the functionality to be executed in an unprivileged level, it increases reliability because there is less code running in privileged mode. Although this model is easy to customise and extend, its performance is affected by the user/kernel boundary crossing. [O'Boyle, 2016]

All the mentioned characteristics and structures were analysed during the development of the project.

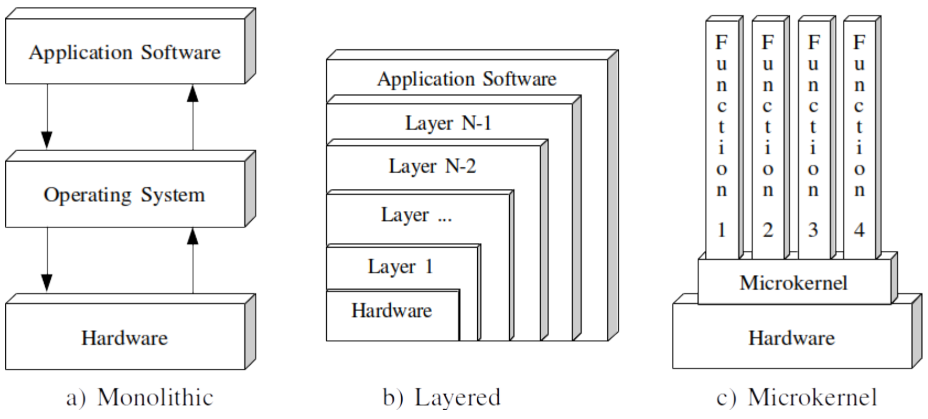


Figure 1: Operating systems design structures

1.3 Hardware

A target hardware device was needed to begin with. A development board, a small portable and cheap computer where to run the product of the project. Previous modules have given students experience working in the ARM environment making it a little easier to start planning the implementation. The amount of documentation, peripherals and other characteristics depend on the specific device chosen to work with was considered before choosing a target computer to start the development.

Product	BeagleBoard	BeagleBone Black	Raspberry Pi 3
Price($)	125	48,87	41,58
Dimensions(cm)	$8,25 \times 8,25$	$8,64 \times 5,33$	$8,5 \times 5,6$

Table 1: Development boards considered [BeagleBoard.org, 2019a], [BeagleBoard.org, 2019b]

For the reasons mentioned above a wide range of boards was looked at, the options that remained after some thought were the BeagleBoard, BeagleBone Black and Raspberry Pi, also taking into account size and price as seen in table 1. From the three boards stated the Raspberry Pi 3 was weighed as the one with more potential and therefore it was chosen for implementation.

1.3.1 Raspberry Pi 3

The Raspberry Pi 3 model B chosen is a single board computer, with the following specifications [Pi, 2019]

- Quad Core 1.2 GHz Broadcom BCM2837 64-bit CPU
- 1GB RAM
- BCM43438 wireless LAN and Bluetooth Low Energy (BLE) on board
- 100 Base Ethernet
- 40-pin extended GPIO
- 4 USB 2 ports
- 4 Pole stereo output and composite video port
- Full size HDMI
- CSI camera port for connecting a Raspberry Pi camera
- DSI display port for connecting a Raspberry Pi touchscreen display
- Micro SD port for loading your operating system and storing data
- Upgraded switched Micro USB power source up to 2.5 A

The most useful characteristics from our perspective for a low-level implementation of an operating system are the processors and the general purpose input output pins, since the approaches taken for the development process and some other characteristics, such as the assembly language used, depend on the processors' architecture, and for communication the easiest way to start with are the pins mentioned.

The Broadcom BCM2837 chip is similar to previous versions of processors in the Raspberry Pi with the difference that it has an ARM Cortex-A53 processor with four cores whose architecture is ARMv8-A. It defines the instruction set used for the coding portion of project. Although these are 64-bit ARM cores, they have full 32-bit compatibility, which makes it feasible to run existing applications developed for 32-bit ARMv7-A.

Testing and debugging were major concerns for all the options considered since it would be impossible to develop an operating system without knowing if it is behaving correctly. As a result an easy way to communicate with the chosen computer, the Raspberry Pi, were the General Purpose Input/Output (GPIO), general purpose input/output, pins. These pins allow us to exchange information without using a keyboard or mouse which would require drivers and which are not existent in a bare metal board without an operating system. The communication established with these pins needed additional hardware to send and receive messages from a host machine and since a better way to do it, such as simulating the implementation, was not found, it was regarded as the easiest way to debug the code, by sending and receiving data from these pins and evaluating the output of those messages.

2 Background

2.1 Operating Systems

Some well-known examples of operating systems are Microsoft Windows with its MS-DOS, macOS and Linux based on Unix, which are developed by quite large groups of people. These are superficially similar but fundamentally different and dominate most of the desktop devices market as shown in figure 2.

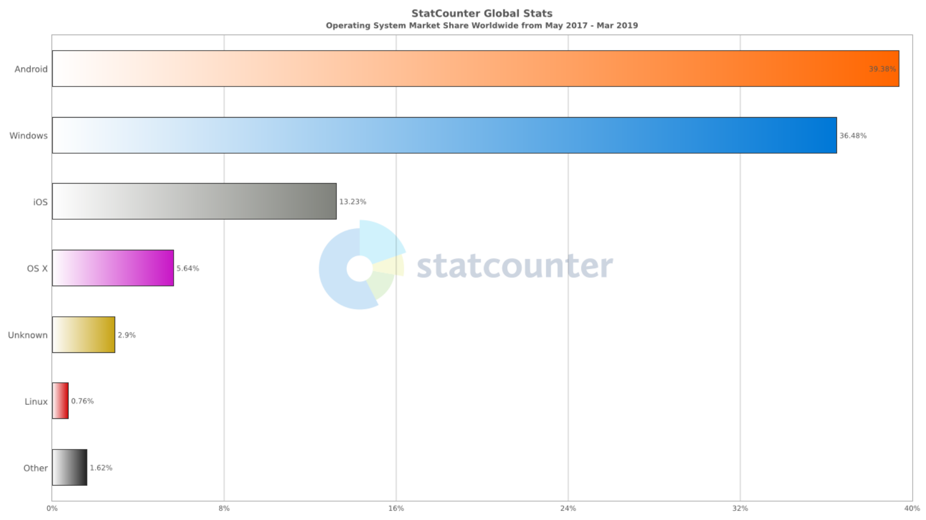


Figure 2: Operating Systems Market Share [Stats, 2019]

Other operating systems are available for mobile devices such as Android and iOS [Adesina, 2014] and have become even more popular than the desktop systems. A small set of systems are more specialised and are used by embedded devices such as Mbed OS.

The mentioned systems share such features as a resource allocator and control program. Resources such as CPU time, memory space, file-storage space, I/O devices, and others are needed to solve problems and these are managed fairly and efficiently by the operating system trying to prevent errors and improper use of the computer. [Silberschatz et al., 2012]

A complete implementation of some of the features needed to manage the resources mentioned was impossible given the time of the project, and for that reason only a few were tried and tested appropriately.

2.2 Functions of an operating system

The most important features considered for implementation and planned to be developed with a little more time are described below.

Security: This protects the resources provided by the hardware with different methods and from a variety of threats which could be mistakes or a serious attacks. A form of security is supported by the hardware architecture which has different modes of operation such as User mode, Operating system mode and Hypervisor mode. [ARM, 2017] If an instruction is executed in a mode without the appropriate permissions a response which could be an error is triggered and then handled. Furthermore, different modes have different sets of registers and are directed to a handler if an exception occurs. Security is also part of memory management since the memory is divided in areas which are only accessible with the appropriate permissions which are defined by the mode of execution; if a restriction is violated then execution branches to a routine to sort out the problem.

Interrupts: These are a form of hardware exception which disrupt the consistent flow of execution of a program getting in between of two contiguous instructions where, depending on the interrupt, the processor usually go back to. There are multiple reasons why an interrupt is raised. A common mechanism is timers which work outside the processor and are triggered for different purposes such as time-sharing scheduling schemes. It is of utmost importance that the interrupted execution can carry on after the system services the request. This is usually handled by a routine running in privileged mode which takes care of the registers and other factors to remain consistent when and after the interrupt occurs. Interrupts can be enabled and disabled with the appropriate privilege by modifying the status register in the processor.

Scheduler: It is fundamental to share processor time among different processes. For general purpose systems the usual approach is to time slice the execution. There is a concept of priority which determines the process in the queue that is going next. In other implementations the priority could indicate the time slice size for each one of them. In time critical systems there is no time slicing because there are not as many things to do and in principle these use a first come first served policy. However, depending on the priority of the process, it can pre-empt another one, replace it and execute before the other has finished. The scheduler makes sure to switch context, keep the state of the process between changes, so nothing fails when it is time to switch back.

Input/Output: A computer with no interaction with the outside world is useless; there must be a way to communicate with a user. There are multiple peripheral devices for input and output, which are managed in different ways and usually require the presence of a driver which is installed once an operating system provides an interface. These devices are managed depending on the hardware available and the application running at a particular moment. There are two ways to handle the communication: one is by polling for an input or output and the other is signalling the processor, requesting to be sorted out, by interrupting.

Memory management: A process needs memory to run which is a limited resource. The space needed depends on the application. Memory can be allocated statically or dynamically; to do it statically we need to know the size to be assigned in advance, when we do not know we modify the quantity while it runs. An ideal way to go through memory is in contiguous blocks since a program will use a determined address range, these blocks are called "segments". It is not always possible to contain an executable within a single block. While segments are being accessed, read or written, the program must not interfere with another program's assigned resources. Therefore, a management procedure is needed, so there is not any harmful disruption in the execution due to unmanaged access. An advantage achieved is that it is possible to present memory as a much bigger resource than it is through memory mapping, abstracting the amount available and storing the contents of segments not in use at the moment to be used by the procedure which needs it.

Filing system: Data sometimes needs to be persistent. Files in memory are lost once the power goes off, and some of them would not fit even if these use the whole memory. There are multiple ways to provide persistent storage and all of them require a structure to access the requested and needed files. The system is managed by filenames rather than numeric addresses which makes it a little more intuitive to work with.

2.3 Characteristics of the operating system implemented

The operating system developed is meant to be simple and easy to understand; its operation on the chosen development board is designed with that objective in mind. As a result, from the few functions of an operating system mentioned, the ones that were considered as the most important to work on during the time assigned for the project were:

- Security: it was implemented by executing the code in the appropriate mode of operation given the specifications of the processor.
- Scheduler: since this is aimed to be a general purpose operating system the approach taken was to develop a time slicing scheduler instead of one based solely on priority.
- Memory management: the memory was allocated statically to make it easier to manage and not having to worry about freeing memory to use for other programs as, these were limited as well.
- Input/output: fundamental to start developing the project as it was the way to see if it was working as expected, test it and debug it.
- Interrupts: needed to manage and access given timers in the Raspberry Pi 3, which helped the implementation of the scheduler, as well as aiding input handlers to help making sure everything worked as expected.

2.4 ARM architecture on the Raspberry Pi 3

The Broadcom BCM2837 chip has an ARM Cortex-A53 processor with four cores whose architecture is ARMv8-A. [ARM, 2014] Each of those cores supports AArch32 and AArch64 execution states, which means they can work with the previous ARM instruction set, the new A64 instruction set and the Thumb instruction set. [Shore, 2014]

> "In the early "pre-ARM" days, ARM stood for Acorn RISC Machines. Then when ARM became a separate company ARM became Advanced RISC Machines and the modern name is just ARM (probably saves people typing!). RISC stands for Reduced Instruction Set Computer which is the type of microprocessor we design." [ARM, 2008]

The difference between the instruction sets is the number of bits they use to encode their instructions and some additional instructions introduced. A64 uses 32 bits, ARM instruction set uses 32 bits and Thumb uses 16 bits for most instructions. The AArch32 and the AArch64 get their names from the width of their datapath, marking a clear difference between the execution states and the instructions sets used.

The instruction sets developed by ARM were renamed after the development of A64, the previous ARM instruction set is now known as A32 and Thumb is known as T16.

The four cores mentioned have thirty one general purpose 64-bit registers plus a program counter, stack pointer and exception link registers. These cores have support for all exception levels, EL0, EL1, EL2, and EL3, in each execution state which are the levels of privilege needed for various operations such as accessing secure memory address space, but only the first two will be used.[ARM, 2014]

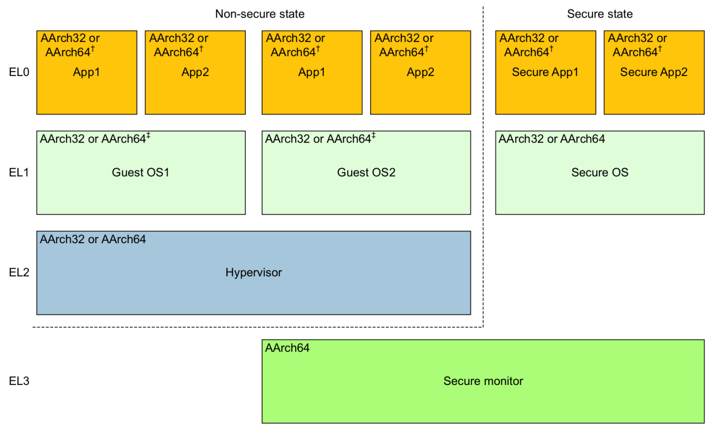


Figure 3: ARM security model ARMv8 [ARM, 2014]

The usual way to use these exception levels is EL0 for applications and EL1 for OS kernel and associated functions that are typically described as “privileged”. The full security model is shown in figure 3

The project was developed in AArch64 execution state and the A64 instruction set.

The Cortex-A53 also has a level 1 cache for each core, which is a volatile fast memory close to the core, with a capacity of 8 kilobytes (kB) for instructions and one of 64kB for data. This is known as Harvard architecture. The four cores share a level 2 cache which has 128kB for instructions and 2 megabytes (MB) for data. The whole structure is shown in figure 4.

The caches were disabled in the operating system developed as it would have made the development process more complicated.

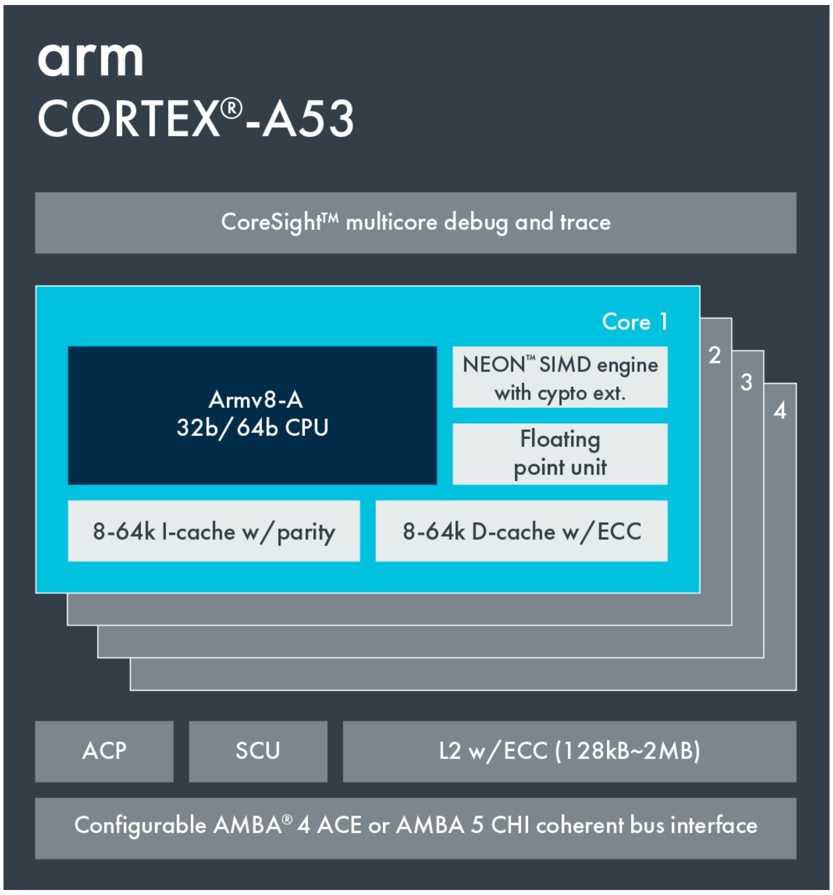

arm
CORTEX®-A53
CoreSight™ multicore debug and trace
Core 1
2
3
4
Armv8-A
32b/64b CPU
NEON™ SIMD engine with cypto ext.
Floating point unit
8-64k I-cache w/parity
8-64k D-cache w/ECC
ACP
SCU
L2 w/ECC (128kB~2MB)
Configurable AMBA® 4 ACE or AMBA 5 CHI coherent bus interface

Figure 4: Cortex-A53 cores’ structure [ARM, 2019]

3 Design and Implementation

3.1 Plan

The initial plan proposed was well structured and it followed what seemed a logical flow. It started by allowing time to research different devices, some of them proposed by the project supervisor and others chosen for their characteristics. Then, familiarising with a subset of the initial devices, trying out some ideas, like finding means of communication between the developer and the hardware without the support of a pre-existent operating system, even checking if extra devices were needed for that purpose. Later, as important as communication is, looking for a way to determine if the execution of the code is as we expected is also fundamental, so tools for debugging, testing or simulating the implementation were investigated as well for some of the devices considered in the subset. Previous experience working with processors serves as evidence that it is better to work with a graphical debugger such as KMD. [Brej, 2019]

After considering the devices, there was the possibility that complementary connectors, wires or some other development products would be needed and the plan allowed for some time to request them before starting with the specific development board chosen. A trial coding stage would take place after all the appropriate materials were in possession of the author. This phase was based on existing programs just to evaluate the complexity of the development process by trying to understand how the code was being load and debugged in the devices.

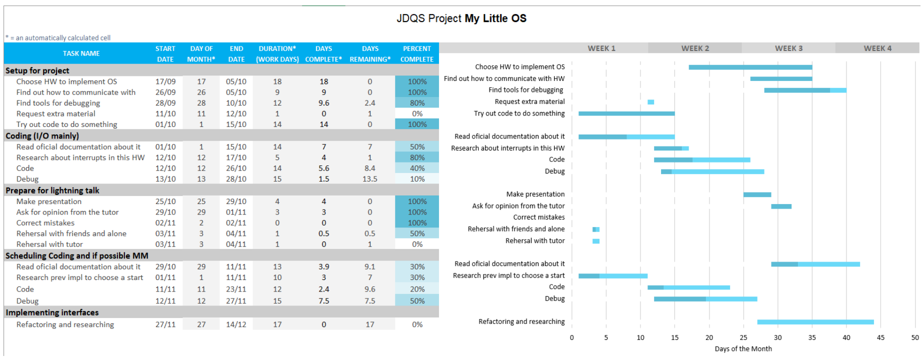
Regarding the input/output and interrupt part it was thought that these topics are closely related and should be coded at the same time. Familiarising with the official documentation and how to find certain things is a valuable skill, especially when there are a lot of pages and documents to refer to. For that reason, some time was assigned to accomplish just that: familiarise the developer with the documentation, especially focusing on input/output handling and interrupts. With that knowledge the development could start, coding a simple function to make sure the behaviour of the input/output and interrupts was understood, not fully but enough to handle basic communication such as receiving characters, probably from a keyboard, and sending them somewhere to manifest in a clear way, such as making a Light-Emitting Diode (LED) flash at different rates.

At this point, the input and output should have been handled and the project was ready to have more functionality with the introduction of a scheduler so it could run more than one application concurrently. More reading would be needed to know how schedulers are usually implemented, especially ones which use the time slicing approach, and what hardware support it needs to run, since the scheduler needs a trigger to switch from one application to another.

Optimistically, memory management was supposed to be introduced in this phase as well, using paging and increasing security by dividing the memory space in segments for privileged and non-privileged access. As a result, an understanding of the memory management unit in the processor was necessary.

The structure was the same as previous stages, read, code and debug the recently created functionality.

Once the mentioned characteristics were coded, debugged and tested, the next thing to do was provide an interface for the applications and the user to communicate with the operating system, maybe even develop a graphical application to manage other applications and interact with the user and other input/output devices which have not been handled yet. It would require more time to design, code and debug. The plan is shown in figure 5.


WEEK 1
WEEK 2
WEEK 3
WEEK 4
0
5
10
15
20
25
30
35
40
45
50
Days of the Month

JDQS Project **My Little OS**

* = an automatically calculated cell

TASK NAME	START DATE	DAY OF MONTH*	END DATE	DURATION* (WORK DAYS)	DAYS COMPLETE*	DAYS REMAINING*	PERCENT COMPLETE
Setup for project							
Choose HW to implement OS	17/09	17	05/10	18	18	0	100%
Find out how to communicate with	26/09	26	05/10	9	9	0	100%
Find tools for debugging	28/09	28	10/10	12	9.6	2.4	80%
Request extra material	11/10	11	12/10	1	0	1	0%
Try out code to do something	01/10	1	15/10	14	14	0	100%
Coding (I/O mainly)							
Read oficial documentation about it	01/10	1	15/10	14	7	7	50%
Research about interrupts in this HW	12/10	12	17/10	5	4	1	80%
Code	12/10	12	26/10	14	5.6	8.4	40%
Debug	13/10	13	28/10	15	1.5	13.5	10%
Prepare for lightning talk							
Make presentation	25/10	25	29/10	4	4	0	100%
Ask for opinion from the tutor	29/10	29	01/11	3	3	0	100%
Correct mistakes	02/11	2	02/11	0	0	0	100%
Rehersal with friends and alone	03/11	3	04/11	1	0.5	0.5	50%
Rehersal with tutor	03/11	3	04/11	1	0	1	0%
Scheduling Coding and if possible MM							
Read oficial documentation about it	29/10	29	11/11	13	3.9	9.1	30%
Research prev impl to choose a start	01/11	1	11/11	10	3	7	30%
Code	11/11	11	23/11	12	2.4	9.6	20%
Debug	12/11	12	27/11	15	7.5	7.5	50%
Implementing interfaces							
Refactoring and researching	27/11	27	14/12	17	0	17	0%

Figure 5: Initial Plan

3.2 Communication

Communication is vital to computer operation. It is essential to find a channel to pass messages to the device, load the programs we are developing, and get a result somehow. In the case of the Raspberry Pi 3 a simple way to achieve this was by using its built-in serial pins to connect to a console and terminal emulation software.

The microSD that comes with the device is loaded with a variety of files concerning configuration and a pre-installed operating system. One of the files contains some configuration for the booting sequence. The file that must be modified is called *config.txt* and a line must be added to it to enable an interface through a serial line. [Bredman, 2014]

In addition a Universal Serial Bus (USB) to TTL serial cable is needed, assuming there is no other serial device to talk to, since it provides connectivity between USB and serial Universal Asynchronous Receiver-Transmitter (UART) interfaces. To use it the host machine requires a driver which comes pre-installed in Linux distributions. The cable is connected to the receiver and transmitter pins of the Pi as shown in figure 6. For our purposes these pins are used as ground, and to transmit and receive data. With the connection made we can initiate the terminal emulator chosen given the parameters needed such as the port where the USB is connected and the baud rate.

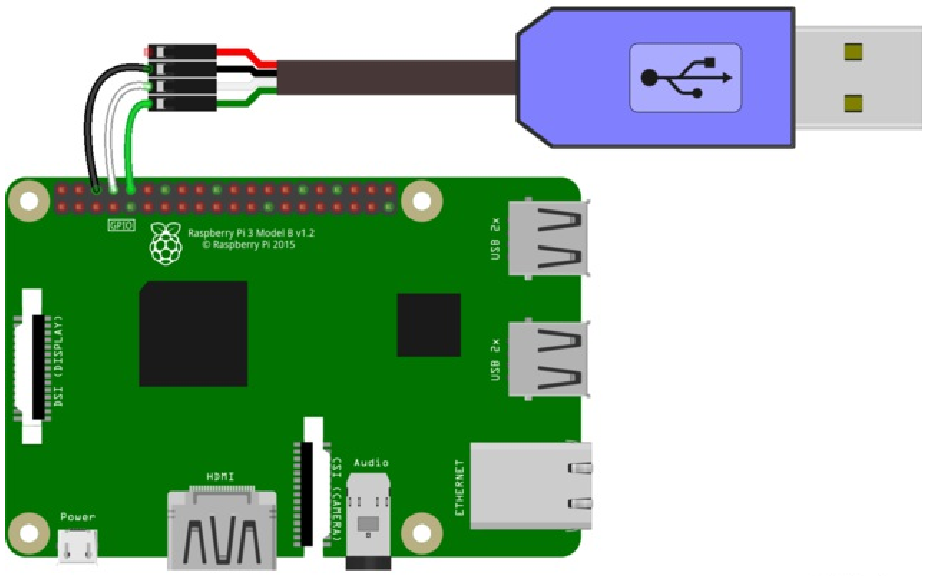


Figure 6: USB to TTL serial cable connection [Monk, 2018]

The process described above is only to make sure we are able to communicate with the device and make sure everything is working properly.

To compile our code a cross compiler is needed, which is responsible for transforming the C and assembly source code into object files. Having the compiled files, the structure of the executable image with its sections is defined. The image obtained is copied to the Raspberry microSD, replacing the one already there and then the execution starts from the beginning of the file.

The boot section starts by identifying the cores with the help of the MPDIR_EL1, which is and identity register, which provides a core identification number. A single core is initially chosen to execute the code which will configure the system by disabling the memory management units, disabling its caches and set endianness for instructions and data. The configuration takes place by modifying bits into the SCTLR register, system control register for exception levels EL0 and EL1 since the level we are interested in is the operating system level, and to degrade the privilege level the execution starts with, everything should be configured.

The last thing we need to be aware of are interrupts which are disabled by modifying the SPSR, Saved Program Status Register. Everything is ready to change our level of privilege from EL3 to EL1, except the address where the application program starts. The address is placed in the Exception Link Register (ELR) of the level the execution is changing from which, in this case, is ELR_EL3, so the address of the instruction to branch to the code to be executed in operating system privilege level is placed in it.

The execution has changed privilege now, but for only one of the cores, while the others have been placed in a low-power standby with the WFI instruction which means the core will wait for an interrupt to resume execution.

3.2.1 Mini UART

The processor is ready to assist us with the mini UART configuration. To begin, the baud rate must be set according to the formula

$$baudrate = \frac{system\_clk\_frequency}{8 * (baudrate\_reg + 1)}$$

[Broadcom, 2012]. The *baud rate* is the baud rate set for the terminal emulator and *baudrate_reg* is the value to be placed in the register which manages the one for the mini UART. Other registers must be configured too. The mini UART provides multiple features including an 8 symbol First-In, First-Out (FIFO) data buffer, which functions as a queue, the first symbol that gets in the queue, will be the first one that leaves the queue. There are status bits to check whether the FIFO is ready to receive or transmit a symbol, and to check if symbols were discarded in case the buffer was full.

With the interface provided, it is easy to create send and receive functions, check if the structure is ready to receive a character, send a character to the FIFO from the host machine, then retrieve the bit which shows if the queue is ready to output the character sent to it. If the data buffers are not ready to either send or receive keep checking the bits which show that information.

With the mini UART and its FIFO buffer working a way of communication has been established, but it is not an efficient implementation because the only thing the system is doing consist of waiting for a character to arrive, polling the status of the buffer, updating it, then checking the FIFO again and since a character is available send said character to the terminal emulator, it is basically "echoing" the message and that is all it can do.

The process of repeatedly checking if certain functionality or resource is available or not is called "polling", which is not recommended as the core is busy largely waiting instead of doing some more meaningful job while the request is ready. A more efficient way to do the same thing is by handling the arrival of characters as an interrupt rather than keep waiting for it.

3.3 Exceptions

In ARMv8 there are 4 classes of disruption in the flow of execution which are:

Synchronous exceptions are caused by the currently executed instruction. For example, trying to store into or load from a nonexistent memory address. In this case, a synchronous exception is generated. A Supervisor call (SVC) is also a synchronous exception: it can be seen as a software interrupt.

Interrupt Request (IRQ) is always asynchronous, which means that it has nothing to do with the currently executed instruction. In contrast to synchronous exceptions, it is generated by external hardware.

Fast Interrupt Request (FIQ) exists for the purpose of prioritising exceptions. It is possible to configure some interrupts as “fast” which means they will be handled first by a separate exception handler which would probably have more resources at its disposal.

System Error (SError), also is asynchronous, and is generated by external hardware. Unlike IRQ or FIQ, SError always indicates some error. For example a non-recoverable error correcting code failure.

Each exception needs a handler. Once an exception is triggered the program counter jumps to a particular location in memory and executes a given routine. The location depends on the exception raised, the memory addresses are contiguous so the system gets what is called a *exception vector table*, which is constrained to a certain space.

The usual approach is to place a branch instruction in each entry on the table and handle the exception elsewhere where there are no constraints of space.

Before servicing the interrupt, the state of the registers must be preserved so the execution can return to where it left off without surprises. In this system the IRQ is the only type of exception that will be handled. An easy way to understand how the process works is by visualising it in figure 7.

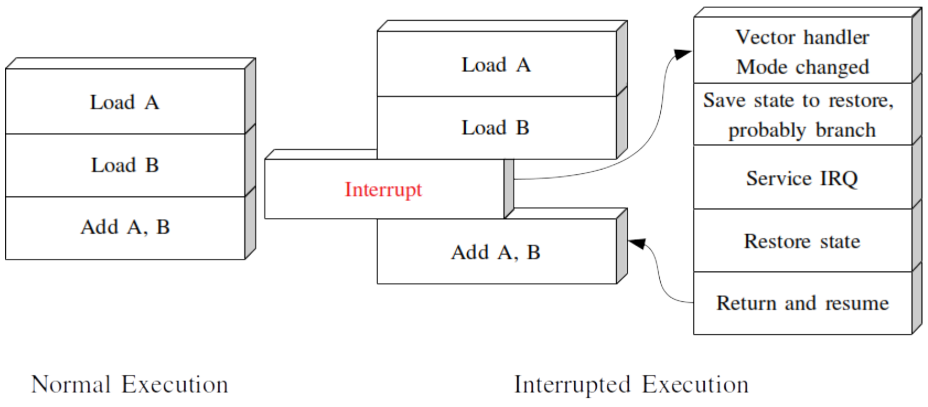


Figure 7: Interrupt handling

3.3.1 Implemented interrupts

There were three interrupts implemented in the operating system developed, two timer interrupts and the mini UART.

If a key is pressed on the keyboard of the host machine it is sent through the mini UART which triggers an interrupt. The job of the interrupt is to retrieve the character from the FIFO and, when it is, notify that the character is ready to be printed.

The first timer interrupt implemented uses a clock inside the processor which is usually running at 1 megahertz (MHz), but it can vary. It means that it is not reliable to measure time.

There is a clock, which is not dependent on the core where the interrupt is handled which makes it appropriate to measure time: it runs at a frequency of 38,4 MHz.

Both timers have a similar interface: they function as counters. There is a register which holds a value to be loaded in another register of the counter which is decreased in every cycle. When the counter reaches zero in the register an interrupt is raised and the register which reached zero is reloaded for the next interruption.

When an interrupt is triggered, the mode of execution raises its privilege to service it. After saving the state of the processor before the handling sequence started, the interrupt vector is examined and control is passed to the addressed handler; subsequently the interrupt is serviced, the state is restored and the privilege is restored to the level it was just before the interrupt sequence appeared, returning to the instruction and state were the execution left off.

3.4 Scheduler

The scheduler is the part of the operating system which manages processor time. It is responsible of providing execution time to each of the processes if these are ready to run. A program is a set of instructions which the computer runs to perform some computation, it requires some information to complete such as variables stored in memory and references to a file system. The instructions, together with the extra characteristics such as who owns the process, the user who started it, is called a “process”. [Breecher, 2018]

A general purpose operating system allows multiple simultaneous processes. These processes usually do not interact between themselves unless it is explicitly stated that they want to cooperate, which means they all have their own space in memory, their own set of variables and registers being used: these are called its “context”. The separation of processes lets the user execute the same program more than once at the same time without any trouble.

Processes in our case are simple, they print a series of five characters using the mini UART interface. The space these process use is statically assigned, since the characters are given when the process is created not at run time. There are two of them for simplicity. To be scheduled the processes must be ready to execute.

When a process is created some code must execute without being switched out, which means it must run without being replaced by something else until it finishes, so this process starts by disabling preemption, which is a temporary interruption in the flow of execution with the intent of resuming later, then it allocates the memory needed for the process to run by assigning a contiguous block of address space, this action can be considered a simple memory mapping, since it is statically assigned and the process will not need more memory while running. After that the current state of the process is set, and a timer interrupt is initialised which is responsible for determining for how long the process will run before it can be preempted. Inside the memory block assigned for the process we include pointers to space where the context of the program can be stored when it is switched out from execution. Then the process is added to the queue of processes to be scheduled.

Regarding the state of processes there are usually three states as seen in figure 8, **ready**, which shows the process can begin execution at any time, **blocked**, when a process is waiting for an event, usually from an input or output device, and **running** which is the process in execution. For simplicity the processes created here do not wait for input or other events so they will only enter the ready and running states.

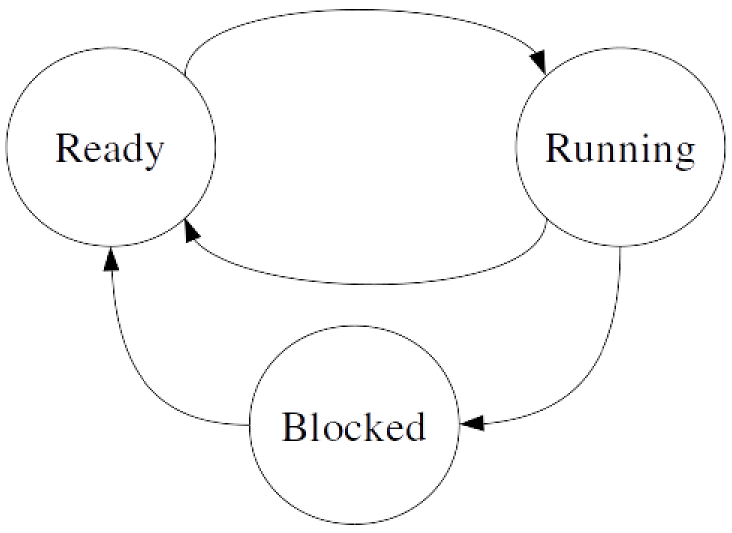


Figure 8: States' flow

The scheduler is triggered by the processor timer interrupt which, as explained before, can vary, for example if the system goes in reduced or low power modes, the timer adapts to the system's performance capabilities, and depends on the processor where we are executing. The job of the scheduler task is to check if the time assigned has finished by comparing the time that has passed with the variable holding the time assigned, which is established as a variable when processes are created. If the process has consumed the time assigned, the variable is reset, the state of the process is changed from *running* to *ready* and its context is stored in the place assigned inside memory space of the process, a process from the queue is picked. Remember, the processes can only be in the *ready* state, and their context is placed in the appropriate registers, then their state is changed to *running* and it continues or starts its execution.

3.5 Debugging

Debugging is essential for every high or low level development. There are multiple ways to debug with the help of tools and devices. The approach taken for this project due to the limited input/output implemented was using the print statement which is one of the most popular techniques used and it is “not as bad as its reputation”, [Clinch, 2017].

The print function was taken from a given implementation, which only needed a pointer to a *putc* routine, which is provided by using the mini UART send function. The library is freely distributed [Nyholm, 2004].

The approach was to follow the flow of execution with print statements which were erased once the functionality was seen to work as expected. It does not provide proof of correctness but it helps find simple mistakes made throughout the process and it may take time to find the error if the print statement is not placed in the appropriate place.

4 Results and Evaluation

The results expected when the project was chosen were optimistic since the complexity of the project was underestimated. Given another chance to pick, the author would have chosen the same one, but more time would have been scheduled even before the beginning of the third year to research the devices to be used in the development process, and also spend some time during summer season reading a little bit of documentation of the device, their characteristics and peripherals in it.

However, the results obtained are not disappointing, and there is a lot more that can be done. The main purpose of the project, which is learning was fulfilled. Time management and version control management are part of the wide range of related skills gained while building the system.

4.1 Plan

The plan does not reflect what happened during the development, but it was a nice guide to follow. The system delivered was supposed to be finished at the end of the first semester, with a couple of weeks to refactor and test the different features implemented.

The mini UART connection took more time than expected and it delayed the development and testing of other functions. It was challenging to make it work. The delays accumulated and the second semester was not used to develop a graphical user interface, but to finish the previous features planned.

4.2 Log

It was indispensable to track the progress made and a logbook was used, it helped keeping ideas fresh since the project time was managed by the author and was not part of the university schedule, and it is hard to familiarise and remember what the developer was doing before taking time to do something else sometimes taking part on the project after weeks, so with a log it was not as challenging to keep working on an idea which was left half way.

The logbook was not as tidy as previous ones, for example the one kept during the Java course in first year, but it was as useful or even more than that one.

Another way to keep track was by using a version control system, the chosen one was GitHub which provided free private repositories at the beginning of the second semester. Each commit was a reference to the feature and its changes. Also it makes it easier to compare differences between versions of the same file to refactor and modify in a much more effective way.

During the first semester most of the development was kept locally and backed up regularly, it was uploaded to a public repository on GitHub but not as often as it should have been, for example each entry on the logbook should have been a commit. As soon as private repositories were available free, the old repository was eliminated.

4.3 Deliverables

A working operating system was implemented which accomplished most of the original objectives presented during the Lightning Talk. It is a simple, well documented operating system which can be used for teaching purposes. However, it does not include a manual or guide besides the comments in the source code as intended and there are multiple features which can be improved.

Despite all its disadvantages a monolithic design approach was used to develop the operating system since it was convenient to have a single structure to manage the interface between applications and hardware because it is not as complex yet.

It was more difficult to refactor and modify features after some time and the number of files became bigger, but the monolithic structure was considered the simplest one to understand since we do not have to worry about the boundaries of the functionality.

Security was delivered by using the different modes of execution available. It is not sophisticated and it has not been tested against malicious programs, but it provides a simple separation between privileged and unprivileged execution. A total of three interrupts were coded and tested, these can be shown through a counter printing on the emulated terminal in the host machine, a key being pressed processed, and the appropriate behaviour of the scheduler.

Multiprogramming was developed as well and it was tested using two simple applications which printed characters in the terminal in a given sequence. This serves as evidence to show that the scheduler process is working as expected. Only one processor is being used so there is no parallelism, the applications run sequentially.

Memory management is involved in the development but it was not completed as it did not include paging or virtualisation and for that reason it was not discussed in the design and implementation section.

4.4 Future

With more time assigned for the project by the author, many topics could be addressed and more functionality provided upon the existing implementation starting with memory management and file-system management before focusing on a user-friendly graphical interface.

Also it would be easier to code since this is a familiar environment to work on now and in retrospective there are some things that could be improved before continuing such as adding a priority field for every process and modify the scheduling algorithm to use it.

5 Conclusions

The experience gained from the project is invaluable. It provides a new perspective on how to tackle problems in different ways when there is not enough time to learn about everything in depth and be comfortable with it, which is absolutely vital in a competitive and ever evolving industrial sector.

The complexity was underestimated since during previous years the author worked in similar laboratory exercises with better suited tools and environment and it did not seem as difficult as the project was, the exercises required a lot of work and sometimes going the "extra mile" was necessary to be satisfied with the results. The hands-on approach made it clear how important and necessary are debugging tools.

The project developed allowed investigation of ideas previously studied but not tried in earlier modules in past years, with some key points mentioned such as scheduling, process interaction, interrupt handling and file system management.

The Raspberry Pi 3 was a sensible choice because there was previous experience working with the ARM assembly language, by doing simple tasks such as system calls handlers, counters, timers and interrupts, so it was not so difficult to adapt to a newer version. However, those tasks were developed with the help of a simulator which made things much easier to spot and as a result correct, and there is not such detailed software implemented for the referenced chip.

The project was a success. It was a challenging task which provided personal and professional experience. There were some problems and surprises which were overcome in a nice and easy way from the perspective of the author, solutions were proposed and tested so in future similar situations a better and quicker decision will be taken which is an important skill to develop.

Technical knowledge was gained regarding ARM assembly and C programming languages. Lots of reading about low level development was required and it refreshed and strengthen the topics learned in previous years.

References

[Adesina, 2014] Adesina, G. (2014). Mobile operating systems and application development platforms: A survey. 6:2195 – 2201.

[ARM, 2008] ARM (2008). ARM Technical Support Knowledge Articles. `http://infocenter.arm.com/help/topic/com.arm.doc.faqs/ka6746.html` (Accessed: 2019-04-20).

[ARM, 2014] ARM (2014). *ARM Cortex-A53 MPCore Processor*. ARM, Cambridge, England CB1 9NJ.

[ARM, 2017] ARM (2017). *ARM Architecture Reference Manual*. Arm limited, Cambridge, England CB1 9NJ, ARMv8-A edition.

[ARM, 2019] ARM (2019). CORTEX-A53. `https://developer.arm.com/-/media/developer/Block%20Diagrams/Cortex-A%20Processor%20Block%20Diagrams/CortexA53.png?revision=f0f8a31a-622d-4ba2-9f14-0b77efb339a1&h=1134&w=1058&la=en&hash=F16F9B2810D168575B84A9C92E50983D6303287C` (Accessed: 2019-04-18).

[BeagleBoard.org, 2019a] BeagleBoard.org, F. (2019a). BeagleBoard. `http://beagleboard.org/beagleboard` (Accessed: 2019-04-10).

[BeagleBoard.org, 2019b] BeagleBoard.org, F. (2019b). BeagleBone Black. `http://beagleboard.org/black` (Accessed: 2019-04-12).

[Bredman, 2014] Bredman (2014). R-Pi configuration file. `https://elinux.org/R-Pi_configuration_file` (Accessed: 2019-04-11).

[Breecher, 2018] Breecher, J. (2018). OPERATING SYSTEMS SCHEDULING. `https://web.cs.wpi.edu/~cs3013/c07/lectures/Section05-Scheduling.pdf` (Accessed:2019-04-19).

[Brej, 2019] Brej, C. (2019). KMD. `http://brej.org/kmd/` (Accessed: 2019-04-17).

[Broadcom, 2012] Broadcom, C. (2012). *BCM2837 ARM Peripherals*. Broadcom Europe Ltd, Cambridge CB4 0WW.

[Clinch, 2017] Clinch, S. (2017). Debugging an Unfamiliar Codebase. `https://online.manchester.ac.uk/bbcswebdav/pid-5720587-dt-content-rid-17796248_1/courses/I3023-COMP-23311-1171-1SE-038538/Workshop3-Debuggingv2.pdf` (Accessed:2019-04-18).

[Garside, 2018] Garside, J. (2018). My Little Operating System. `http://studentnet.cs.manchester.ac.uk/ugt/year3/project/projectbookdetails.php?projectid=34767` (Accessed: 2019-04-17).

[Latham and Garside, 2019] Latham, J. and Garside, J. (2019). M*E*S*H. `https://xerxes.cs.manchester.ac.uk/comp251/` (Accessed: 2019-04-18).

[Monk, 2018] Monk, S. (2018). *Adafruit's Raspberry Pi Lesson 5. Using a Console Cable*. Adafruit Industries.

[Nyholm, 2004] Nyholm, K. (2004). printf.

[O'Boyle, 2016] O'Boyle, M. (2016). Operating Systems Overview. `http://www.inf.ed.ac.uk/teaching/courses/os/slides/00-overview16.pdf` (Accessed:2019-04-11).

[Pi, 2019] Pi, F. R. (2019). Raspberry Pi 3 Model B. `https://www.raspberrypi.org/products/raspberry-pi-3-model-b/` (Accessed: 2019-04-09).

[Shore, 2014] Shore, C. (2014). Porting to 64-bit ARM. `https://community.arm.com/cfs-file/__key/telligent-evolution-components-attachments/01-2142-00-00-00-00-52-01/Porting-to-ARM-64_2D00_bit.pdf` (Accessed: 2019-04-15).

[Silberschatz et al., 2012] Silberschatz, A., Galvin, P. B., and Gagne, G. (2012). *Operating Systems Concepts*. Jhon Wiley & Sons, Inc, Hoboken, NJ.

[Stats, 2019] Stats, S. G. (2019). Operating Systems Market Share Worldwide. `http://gs.statcounter.com/os-market-share#monthly-201705-201903-bar` (Accessed: 2019-04-03).

Glossary

baud rate is the symbol rate, across a transmission medium per time unit using a digitally modulated signal. 15, 17

cross compiler is a compiler that can create executables for a platform other than the one on which the compiler is running. 16

endianness is the order in which bytes are arranged when stored in memory. 16

Harvard is a computer architecture wich has physically separated storage for instructions and data. 13

KMD is a graphical debugger, a software tool for teaching ARM assembly language and interfacing. 14

MPDIR_EL1 is the multiprocessor affinity register, which provides a core identification mechanism. 16

paging is a virtual memory mechanism wich divides memory into pages, determined size memory space, and then addresses them, loads them and translates them. 14, 23

SCTLR is the system control register, which provides a way to set characteristics of the system such as the memory system. 16

SPSR is the saved program status register, it holds the current state of the processor, and it is involved in operations such as enabling or disabling system error interrupts, fast interrupts, etc. 16

Acronyms

ELR Exception Link Register. 16

FIFO First-In, First-Out. 17, 18

FIQ Fast Interrupt Request. 18

GPIO General Purpose Input/Output. 8

IRQ Interrupt Request. 17, 18

kB kilobytes. 13

LED Light-Emitting Diode. 14

MB megabytes. 13

MHz megahertz. 19

SError System Error. 18

SVC Supervisor call. 17

UART Universal Asynchronous Receiver-Transmitter. 15, 17, 18, 19, 21, 22

USB Universal Serial Bus. 15