The University of Manchester
Department of Computer Science
Project Report 2024

My Little Operating System

Author: Thomas H. Jones

Supervisor: Dr James Garside

Abstract

My Little Operating System

Author: Thomas H. Jones

Placeholder

Supervisor: Dr James Garside

Contents

List of Figures

List of Tables

Glossary

ARM A semiconductor design company who design a family of RISC architectures. 6

quantum The time allowed for each process to run before it is paused and another process is ran. 21

RISC-V An open standard reduced instruction set computer architecture developed by the University of California, Berkley. 5, 6, 20, 24, 25

Abbreviations

CLINT core local interruptor

CSRs control and status registers

DTIM data tightly integrated memory

GPIO general purpose input/output controller

IO Input/Output

ISA instruction set architecture

OPT one time programmable memory

PLIC platform level interrupt controller

PMP physical memory protection

PWM pulse width modulator

RISC reduced instruction set computer

SPI serial peripheral interface

UART universal asynchronous reciever/transmitter

Chapter 1

Introduction

There are many low level concepts that are discussed over the course of higher education. This project sets out to explore some these concepts through basic implementation and experimentation, using simple hardware to allow full comprehension of the system being used during development. This chapter will introduce the aims of the project, and describe the motivations behind the project.

1.1 Aims

The overall aim of this project is to produce an operating system for a RISC-V based microcontroller, that allows the execution of multiple processes. To do this, a RISC-V machine must be selected, and a toolchain to develop, load, and execute/debug code on the machine must be made. The elements of the operating system will be split into three main aims, the execution and scheduling of processes, the implementation of basic memory management to restrict each processes memory access, and the development of Input/Output (IO) to allow interaction between the processes and a user. The effectiveness and limitations of the system will then be evaluated, and the RISC-V architecture will be compared with other reduced instruction set computer (RISC) architectures.

1.2 Motivation and Challenges

The basis of this project is to facilitate experimentation with operating system concepts, as while in a university level course the concepts are explained, it is often not possible to showcase these concepts in a practical environment, limited

often to simulations. By using a bare metal system, this project will allow full access to a machine with no other system software, allowing complete control over the environment, which allows the implementation of concepts like processes and memory management.
This has also been used as an opportunity to explore the RISC-V architecture, as the prevalent architecture used in microcontroller systems are ARM based, so the use of RISC-V in this project will enable the comparison of the two architectures, and evaluate the effectiveness of RISC-V as an architecture.
Since the project is being developed on a bare metal system, considerations like the toolchain used to develop and debug code are important as they will have to be configured for the specific system being developed on, which may be more challenging as the userbase of RISC-V is more limited, so the tools required will be smaller and less well documented. The loading of code onto the machine is also important to consider, as it is easily possibly that faulty or buggy builds could cause serious malfunction in the machine, some of which could not be reversible, so the loading and debugging of code must be done with care.
Other challenges include system limitations, as unlike a modern machine that would be used for general use, a microcontroller will have vastly limited resources in comparison, resulting in a need for greater optimization and careful planning, so the requirements of the system do not exceed the resources of the machine. Another limitation is the machine IO capabilities. This limitation does not only affect the end result of the systems IO, but also increases the difficulty of development, as some features such as time based interrupts can be difficult to develop without physical feedback from the machine.

1.3 Report Outline

- Introduction: This chapter gives the basic overview of the planned project
- Background: This chapter will introduce key concepts to enable a more comprehensive understanding of the projects details
- Design: This chapter will outline the key design decisions made before and during the project
- Implementation: This chapter will give details on the practical implementation of concepts and designs mentioned in previous chapters

- Reflections: This chapter will give critical evaluations on the implementation of the project
- Conclusion: This chapter will give a summary of how the aims of the project were met

Chapter 2

Background

This chapter will go over the background of this project, which will cover basic operating system fundamentals, a brief overview of the RISC-V architecture, and the specifics of the board that will be used, which is the Sifive Hifive1 RevB. By the end of the chapter, the reader should be able to understand how parts of the operating system should function, and be able to understand the RISC-V architecture in comparison to the ARM architecture.

2.1 Operating System Fundamentals

An operating system's purpose is to provide an interface between user code and the hardware, such as memory and IO, to allow the user code to function seamlessly, and allow interaction with users. This has been split into three key sections, as listed by the aims of the project, as process management, memory management and IO management.[3]

2.1.1 Process Management

A process is a section of executing code along with the registers that executing code uses. The goal of the operating system is to allow multiple processes to be ran at once. In a system with multiple hardware threads, this would be possible to do explicitly, as the different threads could simple execute the processes. However the desired number of processes is almost always higher than the number of available threads, and in the world of microprocessors there is rarely more than one thread, so instead processes must be run individually and interleaved to give

the illusion of multitasking, provided that this is done at a high enough frequency. There are several methods and approaches to this task. [3]

Scheduling Algorithms

The main approaches to be considered for scheduling algorithms for this project will be a batch approach and an interactive approach. The most important distinction between the two is the inclusion of preemption. In a batch system, a non-preemptive approach will be taken, where once a processes execution is began, it will be allowed to be completed before any other process is ran. This creates a simple system and is used is situations where completion of tasks is not expected to be quick. The main challenges of this approach is in process execution time estimation, as this allows the scheduler to order the execution of processes in a way that ensure small tasks are allowed to run. In an interactive system, it is expected from the user that interactions should have quick responses. This introduces the need for preemption, where a processes' execution may be paused after a short period of time to allow other processes to be ran. By running each process in small interleaved sections this allows multiple processes functionally at the same time, which means processes from which a user is expecting a response will not get halted by a larger process.[3]

2.1.2 Memory Management

The goal of memory management is to streamline how a process can use memory, while at the same time protecting critical sections of memory from faulty or malicious user code. In a system with multiple processes being executed and no memory abstraction, there exists the possibility that two processes attempt to use the same section of memory, creating a conflict that would cause both processes to run incorrectly. This occurs as different processes cannot be aware of each other, and have no choice but to use memory without knowledge of which sections are in use. This can be solved using memory abstraction. The simplest method of this is using address spaces, where each process is given permission to access only segments of directly addressed memory. This prevents each process modifying other processes memory, and allows the process to behave individually, as long as it is provided with the location of its address space.[3]

2.1.3 IO Management

The device that the project features will have severely limited IO. For the purpose of this project there will be two categories of IO, programmed IO and interrupt driven IO.

Programmable IO

Programmable IO is where actions that require IO are done in sequence, which will generally function by the user code making an environment call, which will then perform the IO operation, and return control to the user code once the operation is complete. This is not always desirable, as the operations are often slow and would require polling, which is the process of performing busy operations until a resource is free. During this time all processes are effectively blocked as the the environment call is not being preempted, so this process can significantly slow execution if large amounts of data is needed to be transferred. However this type of IO can be useful in cases where only small amounts of data is being transferred, for instance in cases where only single bytes are transferred the overhead needed for interrupt driven IO would be larger than the time lost to polling.

Interrupt driven IO

To avoid the blocking caused by polling, IO can be interacted with through the use of interrupts. This would be done by a process making an environment call similar to programmable IO, however in this case only the caller process would be blocked until the IO operation was complete. It would also enable some form of interrupt, which signals when the IO function is available to be used. At that point an interrupt will be raised and part of the operation with be completed until the IO is busy again, at which point execution is returned to all the unblocked processes. This means that there is no time wasted polling, although does introduce overhead in the form of interrupt calls, which if done too often could slow down execution to a similar level as polling.

2.2 RISC-V

RISC-V is RISC instruction set architecture (ISA), built with the goal to be completely open. The ISA uses a base integer ISA which can be used on its own or with a number of optional standard extensions, which is part of a goal to avoid

'over-architecting' for particular micro-architectures. There are variants for both 32 bit systems and 64 bit systems however for this project only the 32 bit system will be considered.[5]

2.2.1 Registers

RISC-V implements 32 general purpose registers, which in our case will be 32 bits wide, and labeled x0 to x31. The x0 register is hardwired to zero, and all other registers may be used. There are no specific registers used for storing information like the stack pointer or return address, however in this project the standard calling convention will be used. This specifies how each register should be used, how the register should be saved, and gives each register a pseudonym which correlates to its purpose. Important registers are ra, which is used to store the return address produced by the jump and link instruction, sp and gp, which are used to store the stack pointer and the global pointer. The stack pointer is used to store the most recent item in the current processes stack, and the global pointer is used to point to the address space where a processes global variables will be stored. The collection of registers a0-a7 are used to pass arguments to functions, as well as a0 serving as a return value. For general use there are registers t0 to t6 and s0 to s11, the difference being that t0-t6 are temporary registers that are not required to have their value saved before use, whereas s0-s11 does require saving the value. This allows the use of temporary registers to store intermediate values or value not needed to be stored, without needed the overhead of adding them to the stack, and the saved registers allow important information not to be overwritten by called functions.[5]

2.2.2 Base ISA

The base ISA, known as RV32I, specifies the set of instructions that all RISC-V systems implement, which includes both privileged and unprivileged instructions. The load and store operations allow loading and storing one register to or from memory. This registers can be treated as a word, half word or as a byte. There are the standard arithmetic, logic and shift operations, however immediate values are limited to 12 bits, so operations such as LUI (load upper immediate) may be used to load larger immediate values into a register.
Unlike ARM, the RISC-V ISA does not use flags to determine conditional operations. Instead the condition that is used to determine whether a branch is taken or not is included in the branch operation itself, with instructions like BEQ (branch

if equal) or BGT (branch if greater than). These are accompanied by comparison operations, which evaluate similar conditions, and produce either a 1 or a 0 in the target register. JAL (jump and link) branches to a specified location while storing a return address, which allows the implementation of subroutines and functions.[5]

Privilege levels

The previously described operations can be performed in any privilege mode. RISC-V can support up to three privilege modes, being user, supervisor, and machine mode, although this report will only be considering user mode and machine mode. The zicsr extension adds operations that allow a hart in machine mode to read and modify control and status registers (CSRs), such as the CSRRW, which reads a csr to a register and writes a register to a csr, although either the source or target register can be set to zero to disable either the reading or the writing. Also included is the CSRRS and CSRRC, which is similar to CSRRW but sets or clears bits of the CSRs. Machine mode also has full access to the full memory space, whereas a thread in user mode can only access memory that has been specified by the physical memory protection (PMP), producing an access fault otherwise.[5][4]

Optional Extensions

There are several optional extensions that can be implemented. This includes the 'C' extensions, which includes a reduced number of instructions from the base ISA, but in a 16 bit compressed format. The 'M' extension includes 32 bit instructions to perform multiplication and division, as well as a remainder operation. The 'A' extension includes a number of atomic memory operations, such as store conditional, swap instructions and memory operations that perform logic operations atomically.

2.2.3 Traps

In RISC-V, a trap refers to anything where the execution on the hart is handed to the trap handler. There are two categories of traps, synchronous and asynchronous

Asynchronous

An asynchronous trap is a break in execution caused by external factors, and can be referred to as an interrupt. There are three types of interrupt which are software,

timer, and external. These are controlled by two units, the core local interruptor (CLINT) and the platform level interrupt controller (PLIC). The CLINT handles the software and timer interrupts, and the PLIC handles the external interrupts. A software interrupt is triggered when a hart sets its interrupt bit high. This is used generally for communication between harts, and since this project will not involve more that one hart, this functionality will not be used. A timer interrupt is taken when the mtime CSR is greater than the mtimecmp CSR, so is used to generate an interrupt after a given amount of time. External interrupts are any other caused by the PLIC, which can include separate timer interrupts, or IO sourced interrupts.[2]

Synchronous

A synchronous trap occurs in response to the execution of an instruction, which occurs with the clock, hence synchronous. These come from two sources, errors and environment calls. Errors included memory misalignment's, access faults and illegal instructions. This allows for these errors to handled, either to be inform the process of the error, or to remove the process. Environment calls are how processes make calls to the kernel to perform actions that are above the processes access level, generally to perform IO operations, or to interact with other processes.

2.3 Information on board

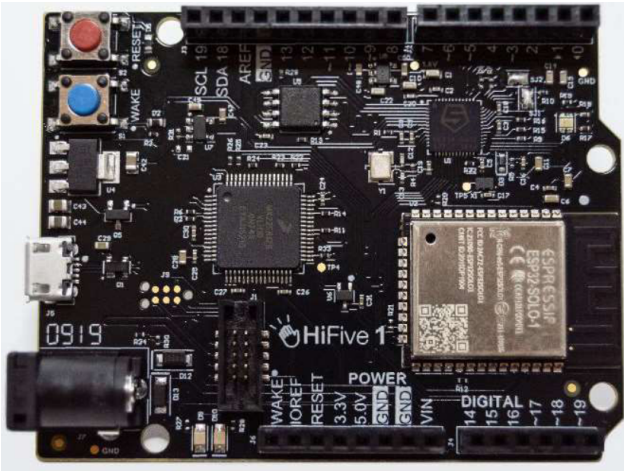

Figure 2.1: This figure shows the Sifive Hifive1 RevB development board from the Hifive1 Rev B Schematics[1]

This project will be done on the Sifive Hifive1 Rev B. This is the second iteration of the Hifive1 board, which uses the SiFive Freedom E310-G002 chip featuring a RV32IMAC core, a 16 KiB instruction cache, 16 KiB data RAM, and 512 Mib of flash memory, which acts as ROM. It has Arduino compatible pins, a set of LEDs, a Bluetooth/WiFi chip, and a USB interface.

Chapter 3

Design and Implementation

3.1 Toolchain

To develop code for the Hifive1, it was required to develop an effective method of compiling and loading code onto the board, as well as an effective debugger. Initially I used the Sifive's Freedom E SDK with PlatformIO to perform basic prototypes and experiments with the board. However these tools did not provide a sufficient level of control over what was compiled and loaded, as the SDK interacts with many parts of system that were needed for the project, as well as altering the configuration of the board. This meant that the SDK was effective for developing simple applications with the board but did not allow for low level control of operations like interrupts or privilege levels. Inspection of source code of these lead to useful insights.
To replace these tools, I used as and ld from the riscv-gnu-toolchain to assemble and link my code into an elf file that could be loaded onto the board. Also used from that collection was objdump, which was used as a disassembler. The assembler's target architecture was rv32ima_zicsr_zifencei, to include all the extensions available on the Hifive1, except for the compressed extension. This was done to prevent the assembler generating a mix of 32 and 16 bit operations, as this causes operations to not be word aligned, and allows jump tables to be implemented without inspecting what instructions are used. Two linker configurations were used throughout the project, the first which put both the text section and the data section into RAM, whereas the second put text and read-only data in ROM, and left the RAM free. The first configuration was used earlier, as it allowed for safe experimentation with the board while maintaining Sifive's double-tap bootloader.

The double-tap bootloader allows normal operation on a regular reset but allows the board to be loaded into a safe mode when the reset is 'double tapped', which loads the board into a safe and known state, so that if the board becomes otherwise inoperable it is still able to be recovered. While this was a useful feature, it limited development, as it also performed several unwanted functions, such as changing the clock configuration, and the uart configuration, which was not acceptable. The second linker config replaced the bootloader in the flash memory with the project code. This was also necessary as it was not practical to store code in RAM, as it was incredibly limited in space, so the RAM was reserved for program data.
To load, run and debug code on the board, I used openocd and gdb. Openocd was used to create an interface with the board, and to specify how the board should be initialized and loaded, and gdb was used to target the openocd interface, which allowed it to load elf files onto the board, and to run/debug code as normal, with some limits such as a limit to the number of breakpoints.

3.2 Board Boot and Configuration

On power on the Hifive1 will begin execution at the reset vector of 0x1004, which in this implementation is not configurable and will jump to the Mask ROM at 0x1_0000, in which similarly cannot be configured in the Hifive1's implementation of the E31. The mask ROM will immediately jump to the one time programmable memory (OPT) memory. This is configurable, but will not be done in this project. This is because the OPT memory can only be edited by code executing on the board, and only one bit at a time. Since the OPT memory is executed as part of the boot sequence, if it is programmed in a way that does not jump to code in flash or memory it will prevent the board from executing code, which also means that the malfunctioning OPT code cannot be fixed, leaving the board in an unrecoverable state. This danger is obviously unacceptable for this project, so the OPT memory will be left to its default, which will jump to 0x2000_0000, which is the beginning of flash memory.
From that point, execution of the system setup begins. On the Hifive1, there are several important configuration options that affect general operation of the board. The most notable of these are the clock settings, as these indicate the frequency of the processor, input and output frequencies, and timer interrupts. This is the second thing configured during boot, after zeroing the registers.

3.2.1 Clock settings

The Hifive1 has 3 clock regions, a high frequency clock, a low frequency clock, and a clock used to drive the JTAG connection. The JTAG driver is constant and only used for debugging through JTAG, so is not relevant here.

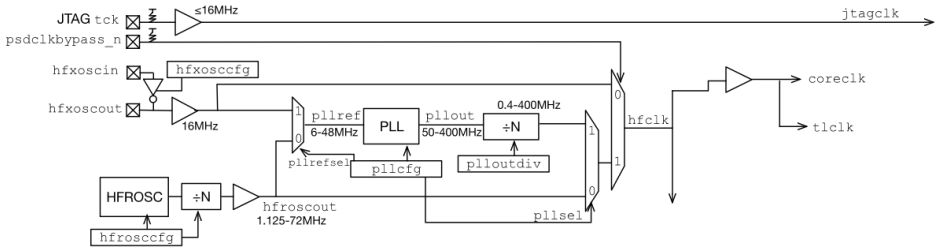

Figure 3.1: The high frequency clock generation scheme, specifying how the high frequency clock is driven and configured, taken from the Sifive FE310-G002 Manual[2]

The high frequency clock controls the processor frequency, and the baud rate of input and output is derived from it. The high frequency clock can be driven from two sources, an internally trimmable high frequency ring oscillator and an external high frequency crystal oscillator. The ring oscillator can produce frequencies ranging form 1 MHz to 75 MHz, whereas the crystal will produce a constant frequency of 16 MHz. Both of these clock sources may be used ‘as is’, or can be modified using a PLL and divider, giving a available range of 48 MHz to 384 MHz.

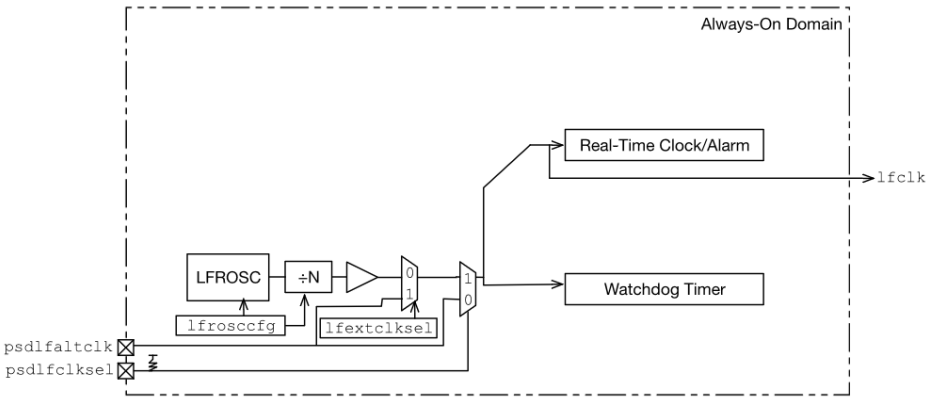

Figure 3.2: The low frequency clock generation scheme, specifying how the low frequency clock is driven and configured[2]

The low frequency clock is part of the Hifive1 'always on block' and controls the watchdog timer, which can be used to cause a reset on malfunction, and both the real-time clock and the machine timer, both of which are used to generate timed interrupts. Similar to the high frequency clock can be driven from a ring oscillator or from an external clock, which in the Hifive1 is a crystal oscillator. The low frequency ring oscillator functions at 1.5 KHz to 230 KHz using a frequency divider, and the implemented external clock runs at a constant 32.768 KHz, with no option to divide the frequency.
For both clock domains, the crystal oscillator was chosen. The ring oscillator gives the option to operate at a higher frequency, which would result in a higher number of operations per second. While in a practical operating system this would be desirable, since this system is not intended for practical use, a constant frequency was more desirable as it would give more predictable results, and makes IO operations more reliable. For the low frequency clock, a high frequency would be beneficial for a real time system as the higher frequency would allow for more precise timing of interrupts and other functions, however for an interactive system this precision is not required, and so similar to the high frequency domain the constant frequency of a crystal oscillator was selected.

3.2.2 Other Settings

After the clock has been set up, various other parts of the system also require a one time configuration, ranging from the trap vector to IO settings.

3.3 Traps

3.3.1 Configuration

When an enabled trap is raised, the pc is set to the trap vector according to the trap mode. The trap vector and mode are stored in the 32 bit csr mtvec. In this csr, bits [1:0] are used to store the mode, where 0 is direct mode and 1 is vectored mode. Bits [31:2] represent vector base address, which is the top 30 bits of a 32 bit address. Since this is stored as 30 bits, two zeros are added to pad the value. This means that the trap handler address must be aligned to 64 bytes. When in direct mode, a trap will set the pc to the base vector address, whereas in vectored mode only synchronous traps are set to the base address, and asynchronous traps set the pc to the base address added to four times the trap code. For this system the vectored mode was chosen. This was because the asynchronous traps are more frequent than synchronous traps, and by jumping to the specific trap call directly a large amount of overhead is skipped. This is relevant for functions like the preemptive scheduler, as it reduces the time to switch processes, which will allow the preemter to be more affective with short quantums.

3.3.2 Asynchronous Traps

Due the use of vectored trap mode, instead of having one single handler for asynchronous traps, each interrupt has it's own handler. Since the Hifive1 does not implement a supervisor mode, there are only 3 available interrupt types, software, timer and external. The software interrupts are not used, as they are mainly used for communication between harts, and the Hifive1 only has a single hart. The machine timer is used for the preemptive scheduler, which will be explored in detail in further sections. Finally the external interrupts, the handler will read from the CLINT claim/complete register. This will provide the handler with the highest priority interrupt code, which will then be used in a jump table to execute the correct interrupt, and once completed the same value is written back to the claim/-complete register to indicate that the interrupt has been handled. The interrupts implemented in this will be explored in further sections.

3.3.3 Synchronous Traps

The trap code for syncronous traps can be fetched from the mcause register. This can then be used with a jump table to handle the correct trap.

3.4 Processes

3.4.1 Process Structure

Due to the extremely limited amount of memory available on the Hifive1, the data required to store information on each process has to be structured carefully, else the amount of memory required to store process information would begin to limit the amount of memory available to the processes themselves. In the current implementation, 35 words are used to store process information.

One byte each is allocated for a process id, process parent id, process status, and size of a processes text section. For process id and process parent id, only one byte is needed as the Hifive1 does not have the memory to support a large of processes, so a theoretical cap of 256 processes is acceptable. Process status can only take 4 values, so only 2 bits of the byte are used, and the text size byte stores the size as a power of 2, where a process has 2^n words, where n is the value stored. The mininum number of bits needed to store these is 13, however this would need to be padded to either 32 bits, to ensure that the structure remained word aligned, as an attempt to use the load word instruction on a non word aligned address causes a load address missaligned error. Instead each entry is stored as a byte, as this is the smallest size load that RISC-V allows, which avoids the need to use bit masks to retieve these entries.

One word each is allocated for the processes program counter, text section pointer and address space pointer. Since memory addresses are word length, these cannot be reduced.

The vast majority of the process structure is used storing the 31 general purpose registers. This is required to retain the state of each process in between scheduling. The only option to reduce this would be to limit the amount of registers available to use. Only 31 must be stored as the x0 register is hardwired to zero.

In other systems, information like the processes stack pointer may be stored, however standard RISC-V calling convention specifies x2 to be used as the stack pointer, so separate storing of this information is not required, and allows a process to handle its address space on its own, however on process creation x2 and x3 are initialized as the stack pointer and the global pointer, where the stack pointer points to the bottom of the process address space and the global pointer at the top. This is done to reduce the overhead of processes that use the standard calling convention.

3.4.2 States

Each process has a state, which indicates how the process should be scheduled. In our model, there are four states that a process can be in, ready, running, blocked, and terminated. Running indicates that the process is currently being exectuted. A running process can transition to ready if the scheduler preempts the process, or if the process yields to the processor with an enviroment call. A process will transition to blocked when it makes an interrupt driven IO call. The blocked process may then transition to ready when the interrupt has been handled. The transition between ready and running is handled by the scheduler, whereas the status of blocked processes are dealt with by each IO handler. The state value stored in each process entry is used as metadata only except for the terminated state, as instead the scheduler and each IO handler tracks state by how each process is stored, for instance any process in a queue to be ran is in the ready state, the current process is in the running state, and any processes not in the queue and not running must be blocked. The terminated state indicates to the scheduler that the process should not be requeued.

3.5 Scheduler

This system will be implemented with an interactive scheduler. The functions that the scheduler must perform are storing all the processes that are in the ready state, running ready processes, and preempting processes that have been running for longer than the quantum.

3.5.1 Creating a Process

On boot, a table of 7 process entries is allocated in memory, and parts of each are initialized, such as each processes address space and their stack and global pointers. The scheduler will then store the pointer to each entry on a stack, which is used to distribute a process entry to each new process. When a new process is created, the pointer on the top of the stack is used to populate that entry with information about the new process, including the start of the text section which is also used as the processes initial pc, the size of the text section, and the process is also given a sequentially generated process id. Once the process is initialized, it is then added to a queue in the scheduler, which is implemented as a circular queue.

3.5.2 Running a process

To make a process run, the scheduler will first dequeue from the ready processes queue. This will then be used to set the processes state. The stored pc will be used to set the MEPC csr, which stores the value of the pc to return to when mret is called. The PMP is then set to allow access to that processes address space, as well as allowing read access to that process text section in the flash memory. The state of the process in the process entry is changed to running. Then most of the registers have their values set, except for t1-4. This is because before these values are set, the sheduler sets the quantum for the process by setting the timer comparison to the current time plus the quantum, and this requires a mininum of 4 registers to to. This is done as close to the start of execution instead of before the registers have their state set, as this ensures the time allowed for each process is as close to the specified quantum as possible. Finally the last 4 registers are set and MRET is called, which begins the execution of the process. The operations of

3.5.3 Preemption

Each process is given 500 milliseconds of execution time. This time includes execution of user code as well as enviroment calls, however a process cannot be preempted in the middle of an environment call, so if a process reaches the end of its execution time during an environment call, it will be preempted as soon as the system returns to the user code. As previously mentioned, the preemption is timed using the machine timer interrupts. The machine timer was chosen over the timer interrupt generated by the AON unit, as the latter interrupt is part of the external interrupts so there is an increased amount of overhead required for it to be processed. Using the machine timer interrupt also addresses another problem, which is maintaining the state of the process. When the preemption occurs, the first thing that must happen is that the processes registers must be saved to the process entry, which requires the global pointer. The global pointer is a constant value, as it always points to top of the machine address space, however to use it, it must be stored into a register, which requires that registers value to be store to the machine stack. For this we use the mscratch csr to store the machine pointer during normal execution, which when in machine mode we swap with the current processes stack pointer. This allows us to use the machine stack while also storing the processes stack. In addition to the global pointer we also save the value in t0 and t1 to store the process entry pointer, and to pop and store the values back from the stack and the mscratch csr, before saving each of the remaining 26 registers.

The operations of the preempter occur as follows:

1. The PMP is reset, clearing all user memory access
2. The fence.i instruction is called, which invalidates all entries in the instuction cache
3. The process sp is switched with the machine sp in the mscratch csr
4. Registers gp, t0, and t1 are pushed to the stack
5. The global pointer is loaded into gp, and used to load the current process entry pointer into t0
6. The 3 values pushed to the stack are each pop into t1 and then stored into the process entry, as well the the value that was swapped into the mscratch csr
7. All 26 remaining registers are stored to the process entry
8. The value stored in the mepc csr is loaded and stored
9. If the process had the running state, it is changed to ready, otherwise it remains unchanged.
10. If the status is terminated, add the process entry pointer to the free entry stack
11. Else add the pointer to the process queue

Blocking and Yielding

Scheduling ready processes and terminated processes is intuitive, however when a process is blocked, the scheduler is instructed to ignore that process. This is because it is the job of the blocking source to store its blocked processes. This reduces the comlexity of the scheduler and gives more freedom as to how and when a process in unblocked.
When a process is yields or is blocked, they effetively request immediate preemption. One method of doing this would be to call the scheduler directly from whatever enviroment call or interrupt caused the yield or block. However this could create an inconsisten state, as the scheduler preempter is always called from

an interrupt to user code. Because of this, when a process yields or is blocked, it simply zeros the timer compare, so that when it returns to the user code it will immediately raise an exception, allowing the scheduler to handle the process as normal.

3.6 Memory Management

3.6.1 Address Spaces

For this project, RISC-V's PMP will be used to implement and enforce address spaces. The Hifive1 has two forms of memory, a 512 MiB flash memory to serve as ROM and an 16 KiB L1 data tightly integrated memory (DTIM) which served as RAM. Both of these sections allow for execution of code. Because of this and the limited amount of RAM, the text section of each program will be stored on and executed from the flash memory. The 16KiB of RAM is split into 8 address spaces, making each space 2KiB. This size was chosen as a balance between number of available processes and amount of memory available to each process, as well as ensuring that the space needed to store information of the processes did not take up the majority of the machine address space. This is because since each process entry requires 35 words or 140 bytes, so if split into 8 address spaces there will be 7 available processes, which would take 980 bytes. So with only 7 processes, the process entry table would already take 48% of the machine address space. This could be adjusted by keeping the machine address space at 2KiB while decreasing the address space size to each process, however it is useful to have the address spaces be in powers of 2, as will be discussed later with the PMP. If equal sized spaces were maintained but split into 16 1 KiB spaces, the 15 process entries would need 2100 bytes, which is over 200% of what is available. If we were to increase the size of the spaces to 4 KiB, that would only make 4 address spaces, with only 3 available processes, which is too small to be practical.

3.6.2 Address Space Layout

Each 2 KiB address space has 2 pointers. The global pointer, gp, points to the bottom of the address space, and is used to store global variables for a process, so should not change unless a program chooses to use the gp register for another purpose. The stack pointer, sp, initially points to the byte above the top of the address space. This is used to implement a full descending stack. Sp will always

store the pointer to the last item added to the stack, so when the stack is empty it will point to the byte outside of the stack.

3.6.3 PMP Use

To enforce the address spaces, RISC-V's PMP is used. PMP allows us to specify sections of code to have have restricted access to calls from user mode. This is done by setting the values in the pmpaddr and pmpcfg CSRs. It can be done in two ways, either the top of range setting which protects the region between a specified top address and either the previously referenced address, or 0x00000000. This allows a precise range however it would require storing two values in each process for its text section. The other method is to specify an address which is naturally aligned to a power of two. This means that the address is a multiple of that power of two. The value of the power of two is specified by using the trailing zero bits of the naturally aligned address, where all but the most significant zero bit is set to one. The configuration for each is stored as one byte, with 5 fields, R, W, X, A, and L. R, X, and W control the read, write and execution access in each region, A controls whether that control region is active and the method which the region is specified, and L will lock the settings and also enforce the settings to both user mode and machine mode until reset. When L is not set, code executed in machine mode will ignore the PMP settings.
These addresses are store in the CSRs pmpaddr0 to pmpaddr7, which means that the Hifive1 implements up to 8 PMP controlled regions. The configuration of each of these addresess are store in the CSRs pmpcfg0 and pmpcfg1, which each control pmpaddr0-3 and pmpaddr4-7, respectively.
Since the address spaces in RAM are constant the pmpaddr value can be generated by setting address[8] low and address[7:0] high. The text section of a process has a variable size, which is stored as the value n, where the size of the section is 2^n words. There are then two cases to consider when implementing this. When $n == 0$ it represents the case when the size of the section is only one word. Since this will always result in a naturally aligned address, there is a seperate value of A to represent this NA4 case, so all that is required for this case is to set A to 2, and specify the word address. The general case where $n > 0$ works as previously specified.

```
1       # Calculate the pmpaddr value for the text section
2       lw t1, p_text_pointer(t0)
3       # Value is shifted 2 bits right as it is stored as
            bit [33:2] of the addr
4       srli t1, t1, 2
5       lb t2, p_text_size(t0)
6       # In the case where n == 0 or n == 1, no
            alterations to the address occur, so jumps
7       li t3, 1
8       ble t2, t3, set_pmp_addr0
9       # pmp_set_high = 0x01ff_ffff
10      li t3, pmp_set_high
11      # max_text_size = 26
12      li t5, max_text_size
13      sub t5, t5, t2
14      srl t3, t3, t5
15      or t1, t1, t3
16  set_pmp_addr0:
17      csrw pmpaddr0, t1
18      # Data address spaces are a constant size
19      lw t1, p_data_pointer(t0)
20      srli t1, t1, 2
21      andi t1, t1, ~0x100
22      ori t1, t1, 0xff
23      csrw pmpaddr1, t1
24      #set pmp1cfg, data value is constant
25      li t2, pmp_data
26      # If text size is 0, A=2(na4)
27      # Otherwise A=3(napot)
28      lb t1, p_text_size(t0)
29      beq t1, zero, set_na4
30      ori t2, t2, pmp_text_napot
31      j set_pmp_cfg
32  set_na4:
33      ori t2, t2, pmp_text_na4
34  set_pmp_cfg:
35      csrw pmpcfg0, t2
```

Figure 3.3: Set up of the PMP when a process is scheduled, from ustil.S

3.7 IO

The Hifive1 has few forms of IO available. Most forms of IO are driven through the general purpose input/output controller (GPIO). This controls the function of each of the arduino compatible pins, as well as up to two alternate functions for each GPIO pin, which is used to control the universal asynchronous reciever/-transmitter (UART), serial peripheral interface (SPI), and pulse width modulator (PWM), as well as the RGB leds featured on the board.

3.7.1 LEDs

The simplest form of IO on the Hifive1 are the three leds, which can be used as simple indicators, however they utilise the GPIO just as the other IO functions do. The green, red, and blue LEDs use GPIO pins 19, 21, and 22 respectively. To use them, the bits 19, 21, and 22 in the GPIO output enable register, memory mapped to 0x1001_2008, must be set low, since the GPIO uses low enable. This enables the LEDs to function, which can turned on and off by setting the bits in the GPIO output register similarly to how they were set for the enable register.

3.7.2 Text input and output

The main way a user may interact with a program in this system will be through the use of a terminal, which requires the need to print and recieve text. To do this, we will utilise one of the two UART peripherals. This will allow us to send and recieve text from a host machine that is connected via the USB port.

Host machine

When the Hifive1 is connected to a host machine, the two UART peripherals are represented as two ttyACM devices, which can be interfaced with using the two files /dev/ttyACM0 and /dev/ttyACM1. To fetch and send data, the terminal used must be configued to function at the target baud rate, which can be done using stty or through other applications settings.

UART configuration

To enable the use of UART, the specific peripheral must be enabled through the GPIO. To do, bits 16 and 17 of the iof_en register must be set to 1, and same bits

of iof_sel should be set to 0. This toggles GPIO_16 and GPIO_17 from software control to hardware control from the UART0 peripheral. After this, UART0 must be configured. The first setting to alter is the frequency divider, which is used to determine the baud rate of the peripheral from the high frequency clock. This uses the formula

$$f_{baud} = \frac{f_{hfclk}}{div + 1}$$

. The target baud rate will be 115200 Hz. Using the hfclock frequency of 16 MHz, we can deduce that the required divisor will be 138, which give us the actual baud rate of 115107 Hz, which has an error 0.08%. Other settings include setting the rx_en and tx_en to high, to enable both reading and transmitting. Other configurations include the watermark level, which will be discussed in a later section.

Transmission

The transmission of data from user is interrupt driven. To begin a transaction, a process will make a enviroment call, passing it a pointer to the first character of a zero terminated string as an argument. First the calling process is blocked, and that process is queued onto a queue structure, which stores all processes that are currently blocked due to an ongoing request to transmit. The string pointer is then queued onto a second queue, with all the processes strings. It then checks if the transmission interrupts are already enabled due to other requests, and enables them if not, before returning from the enviroment call.

Recieving

Chapter 4

Reflections

4.1 Design Reflections

Chapter 5

Conclusions and further work

Bibliography

[1] *Hifive1 Rev B Schematics*. Sifive Inc, 2021.

[2] *SiFive FE310-G002 Manual v1p5*. Sifive Inc, 2022.

[3] H. B. Andrew S. Tanenbaum. *Modern Operating Systems 4th Edition*. Pearson, 2015.

[4] J. H. Andrew Waterman, Krste Asanovi, editor. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 2021120*. RISC-V International, December 2021.

[5] K. A. Andrew Waterman, editor. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2019121*. RISC-V Foundation, December 2019.