Porting the Unix Kernel

Christopher K. Hettrick

University of Victoria
Department of Computer Science
CSC490
Supervised by Dr. Bill Bird

ABSTRACT

This report describes the process of porting a variant of the Unix kernel from the MIPS architecture to the Arm architecture. A heavily modified 2.11BSD version of the Unix kernel called RetroBSD is used as a case study, and is the basis of this development. The goal of this project is to run this ported kernel on both an emulator and on a physical embedded development board. An additional portion of this work is devoted to adapting the large-scale codebase of RetroBSD to more modern and sustainable development standards that will facilitate future ports to other platforms and architectures. The host development environment is supported on both OpenBSD and Linux as host operating systems for cross compilation to MIPS and Arm targets.

27 December 2020

Table of Contents

Porting the Unix Kernel

Christopher K. Hettrick

University of Victoria
Department of Computer Science
CSC490
Supervised by Dr. Bill Bird

1. Introduction

Porting the MIPS32® M4K® architecture to the Arm® Cortex®-M4 architecture.

XXX Words

XXX Words

XXX Words

XXX Words

XXX Words

2. Relevant History of Unix and BSD

RetroBSD is a semi-modernized version of 2.11BSD targeted to the PIC32MX7 MIPS-based microcontroller.[1] The early history of RetroBSD has been lost. It can only be concluded that RetroBSD was started some time in 2011, or perhaps some time even before that. The earliest post on the RetroBSD forum was from August 15, 2011. The project could have started much earlier than the creation of the forum. The project was started and lead by Serge Vakulenko, a systems programmer who started working at MIPS Technologies in 2011.[2]

2BSD is a family of operating systems for the DEC™ PDP-11 derived from Research UNIX and developed at the University of California at Berkeley. 2.11BSD has a long lineage going back to the first release of 2BSD on May 10, 1979.[3] 2BSD is a direct descendant of the Sixth Edition of Research UNIX, commonly known as V6 UNIX. 2.8BSD incorporated features from the Seventh Edition of Research UNIX, 32V UNIX, and 4.1BSD. The 2BSD line of software distributions continued on until the most recent release of 2.11BSD in 1991.[4] This release was a celebration of the 20th anniversary of the PDP-11. It is the culmination of the many efforts to port features from 4.3BSD and 4.3BSD-Tahoe — which run on the DEC VAX — to the PDP-11. Patches to 2.11BSD have been sporadically available since the initial release in 1991 from the long-time maintainer Steven Schultz. The most recent patch level is 469 and was released on April 14, 2020.[5] RetroBSD was started from patch level 431, which was released on April 21, 2000. It is from this version that all RetroBSD development began.

DiscoBSD derives from the most recent commit to the RetroBSD codebase, which is revision 506 from February 17, 2019.[6]

3. Hardware

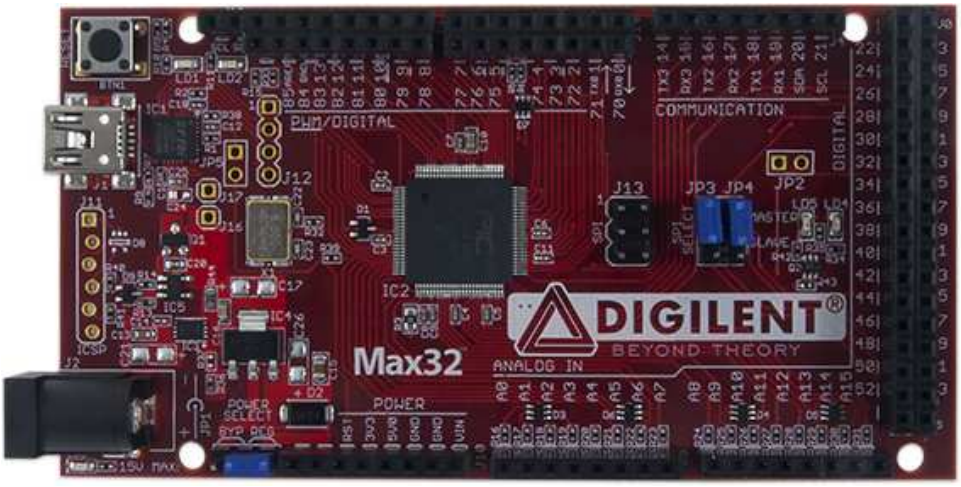
The defining features of the target hardware for RetroBSD and DiscoBSD are that they are RAM-constrained, have 32-bit processors, and do not have a memory management unit (MMU). The lack of an MMU rules out any possibility of virtual memory, which is a critical component in most major operating systems. A secondary feature of the target hardware is that their processors have the ability to protect kernel code from user code with a memory protection unit. This feature was not explored in this project, but is a viable focus of additional study.

3.1. PIC32 Development Board

The default development board that was used for the design and development of RetroBSD is the Max32 board, produced by Digilent. It employs a PIC32MX795F512 32-bit MIPS-based microcontroller. The processor runs at 80 MHz, has 512KB of flash program memory, and 128KB of SRAM data memory. It is powered either through the external power connector or the onboard USB connector. A UART terminal is achieved through the USB connection. This board does not have an onboard SD card, but one can be made available through an external Arduino-compatible shield. The SD card connects to the SPI2 port of the microcontroller. 83 general purpose I/O (GPIO) pins and some onboard LEDs are available. The board offers many more peripherals than outlined here, but they have no relevance to this project.

Programming and debugging of the board is achieved through the use of the PICkit3 in-system programmer/debugger and the connected USB cable. The MPLAB IDE software is used on a Windows system to load firmware into the flash memory of the microcontroller.

An image of the Max32 development board is shown below:



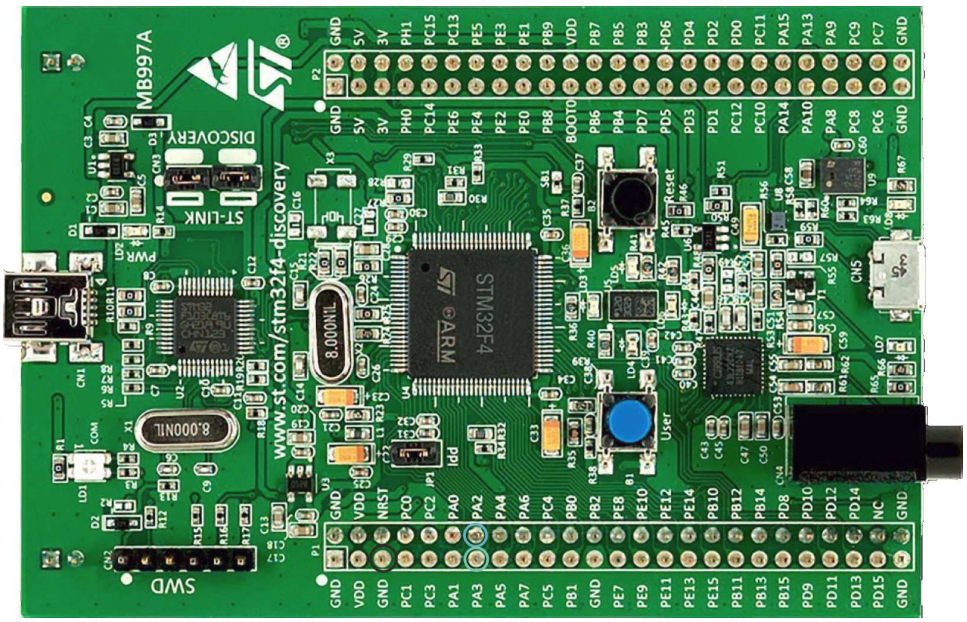
3.2. STM32 Development Board

The default development board that is used for the design and development of DiscoBSD is the STM32F4Discovery board, produced by STMicroelectronics. A fully compatible revised edition of the board has been released under the model name STM32F407G-DISC1. All revisions of the boards employ an STM32F407VGT6 32-bit Arm Cortex-M4 microcontroller with a single-precision floating-point unit. The processor runs at a max speed of 168 MHz, has 1MB of flash program memory, and 192KB of SRAM data memory. It is powered through the onboard USB connector. A UART terminal is achieved through the USB connection. This board does not have an onboard SD card, but one can be made available through a connection to the I/O pins on the extension headers. The SD card connects to one of the microcontroller's SPI ports. 80 general purpose I/O (GPIO) pins, one User button, and four onboard LEDs are available for use. The board offers many more peripherals than outlined here, but they have no relevance to this project.

A unique aspect of this microcontroller is that 64KB of its 192KB of SRAM is Core Coupled Memory (CCM), which is only available to the processor's data bus. This area of RAM is suitable for kernel data structures, as it does not go through the Multi-AHB bus matrix, thus has zero wait states and no contention for bus access. SRAM1 has 112KB of the total 192KB of SRAM, which is suitable for either user or kernel processes. SRAM2 has the remaining 16KB of the total 192KB of SRAM, which is tied to the System-bus and only available for peripheral data transfers, i.e., USB, ethernet, or DMA data transfers. The SRAM2 is not used in the DiscoBSD kernel.

Programming and debugging of the board is achieved through the use of the onboard ST-LINK/V2-A in-system programmer/debugger and the connected USB cable. The STLink software is used to load firmware, such as the DiscoBSD kernel, into the flash memory of the microcontroller. Remote debugging of firmware running on the physical board is possible through the GNU debugger, used as a remote protocol server, paired with the OpenOCD on-chip debugger to access on-chip debug facilities. The project's top-level Makefile has targets for running OpenOCD in connection with the GNU debugger.

An image of the STM32F4Discovery development board is shown below:



4. Simulators and Emulators

Employing either a simulator or an emulator for the development of an embedded system is an efficient use of limited resources and reduces unproductive time during the code-compile-load-debug development cycle. They are also valuable in enabling system development during a lack of availability, or access, to the physical hardware.

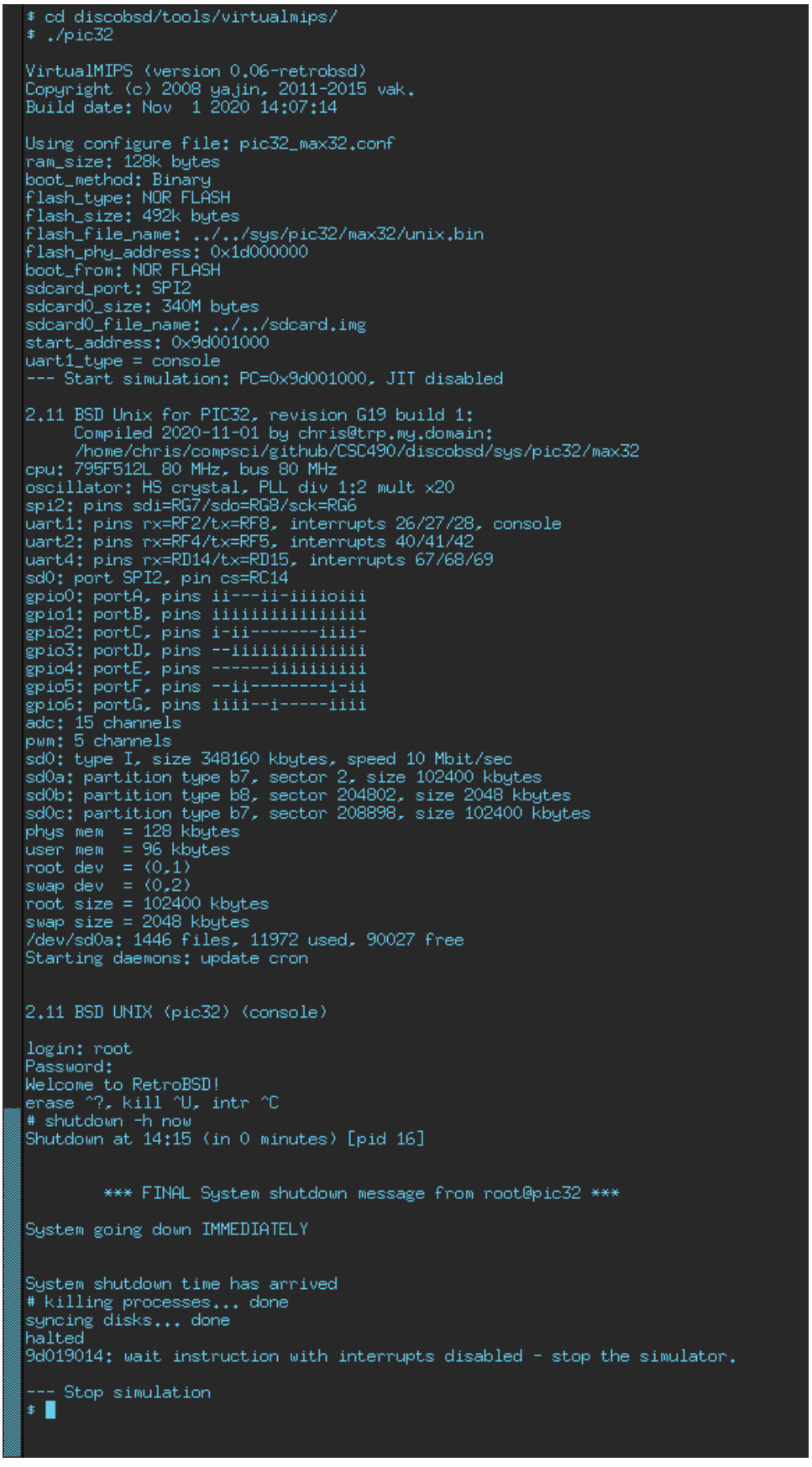
Simulators and emulators are cycle-accurate representations of the physical hardware. They are computer programs that offer the same processor and common peripherals as available on target development boards. The development process amounts to loading a compiled binary firmware file, an *Intel Hex* formatted file, or an *ELF* formatted file into the simulator or emulator. A debugger, such as the GNU debugger, is attached and used to run and interrogate the system-under-test.

4.1. PIC32 VirtualMIPS Simulator

The VirtualMIPS simulator is used to boot and run the RetroBSD MIPS-based kernel and userland. It is bundled with the RetroBSD codebase, and is available in the `/tools/virtualmips` directory. The simulator executable is named `pic32`. RetroBSD compiles separate kernels for each of the various PIC32-based development boards. By default, VirtualMIPS is configured to simulate a Digilent Max32 board and runs the `/sys/pic32/max32` kernel along with the common MIPS-based userland. The kernel, named `unix.bin`, is provided as a binary firmware file. The simulator provides virtual peripheral devices such as an SPI port for the SD card interface, a UART for the console terminal, and GPIO pins for toggling LEDs. Pulse width modulation and analog to digital coverter peripherals are also simulated.

VirtualMIPS compiles and runs on Mac OSX, OpenBSD, and Linux, although only OpenBSD and Linux have been tested. Debugging a RetroBSD kernel with the GNU debugger through VirtualMIPS was not attempted, but by all indications it is possible, as the developers of RetroBSD debugged and developed in this manner on Mac OSX and Linux.

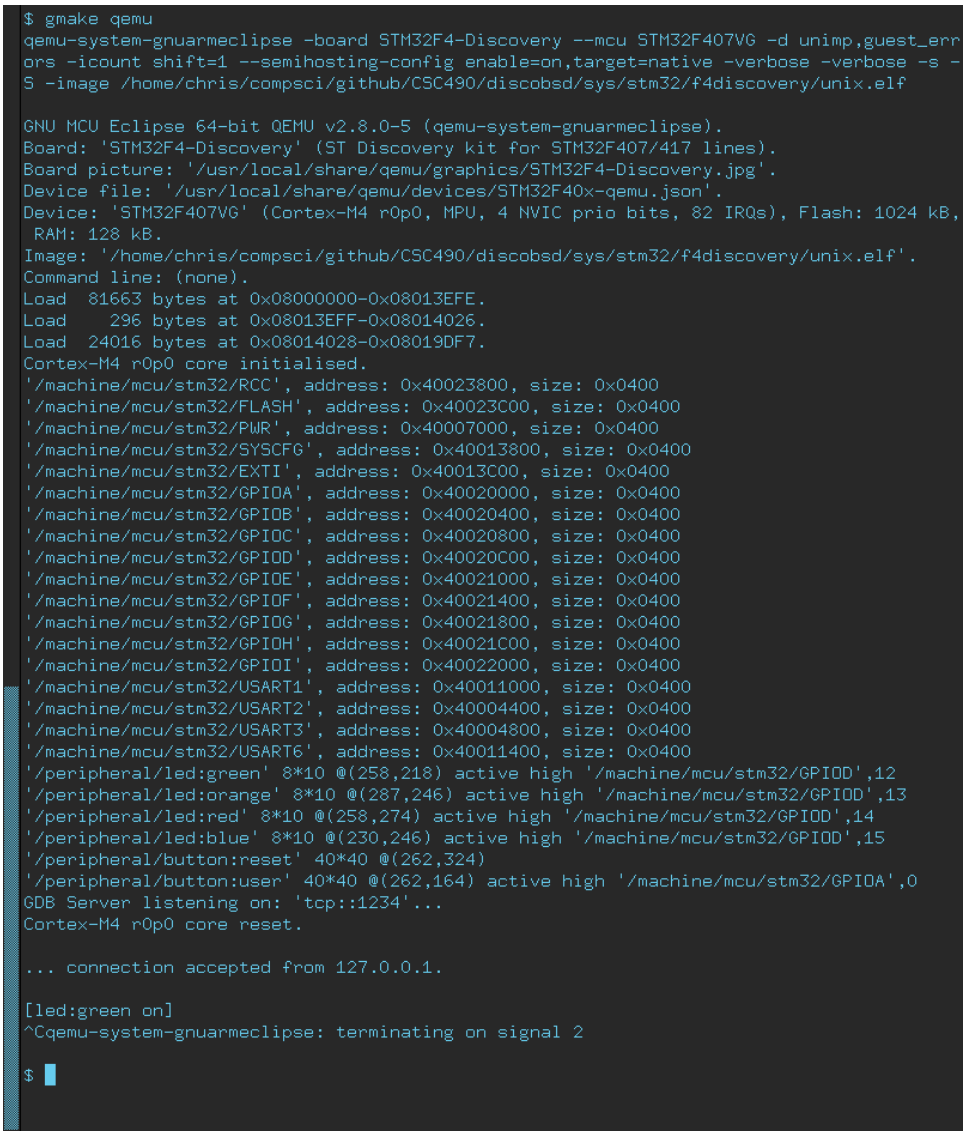
An image of RetroBSD booting in the VirtualMIPS simulator is shown below:



4.2. QEMU-based Arm Cortex-M Emulator

The QEMU-based Arm Cortex-M emulator (hereafter called qemu-arm) is used to boot and run the DiscoBSD Arm-based kernel and userland. It is available through the various package managers on Linux and as a custom user-compiled port on OpenBSD. `qemu-system-gnuarmeclipse` is the name of the emulator executable. DiscoBSD currently compiles a single kernel, targeting the STM32F4-Discovery development board. qemu-arm is configured on the command line to emulate a STMicroelectronics STM32F4-Discovery board and run the `/sys/stm32/f4discovery` kernel along with the Arm-based userland. Note that the Arm-based userland is not yet complete. The kernel, named `unix.elf`, is provided as an *ELF* formatted firmware file. The emulator provides virtual peripheral devices such as a USART for the console terminal, and GPIO pins for toggling the four onboard LEDs and reading from the user button. The many other services afforded by the standard QEMU are present in qemu-arm but have not been explored in this project.

An image of DiscoBSD booting in the qemu-arm emulator is shown below:



The qemu-arm emulator runs on OpenBSD and Linux, although only OpenBSD has been used for development. Debugging a DiscoBSD kernel running on qemu-arm with the GNU debugger is possible and is integrated into the codebase. The project's top-level Makefile has targets for both running qemu-arm and running the GNU debugger with qemu-arm as the target.

5. Host Development Environment

This project was developed on Unix-based host operating systems. Development was mainly on the OpenBSD operating system, while compatability and portability testing was performed on Linux as a host. The original RetroBSD project was developed on Mac OSX and Linux, with support for FreeBSD as a host near the end of RetroBSD's timeline.

The DiscoBSD host development environment consists of a number of main development tools:

- a binary flash downloader
- a circuit board simulator or emulator
- a compiler, assembler, and C library
- a source-level debugger
- an on-chip debugger

As supporting tools, these commonly-present Unix programs are also required:
awk, bison, byacc, gmake, sed, shell (either Bourne or Bash)

The host development environment created for this project is targeted to build and develop for both the MIPS-based RetroBSD kernel and the Arm-based DiscoBSD kernel. A specific aim of this project is for the codebase to concurrently support many architectures, starting with the original MIPS code and then expanded with the new Arm code developed for this project as DiscoBSD.

5.1. Development Tools on OpenBSD

Significant resources were allocated to the construction of a suitable development environment for both MIPS and Arm targets on a Unix-based operating system, as an alternative and addition to the well-established Linux operating system. OpenBSD was chosen for this task, as it is dissimilar in many ways to Linux, while still maintaining Posix compliance. This satisfies an aim of this project for the development and testing of portability between various host development systems.

Using the OpenBSD Ports Collection, custom user-compiled ports of third-party software was developed. These include the mips-elf targeted GCC toolchain, the STLink binary flash downloader for STM32 devices, and the qemu-arm circuit board emulator. The remaining software packages needed for development are available as an OpenBSD package through the pkg_add system. These packages include the OpenOCD on-chip debugger, and the Linaro version of GCC targeted to arm-none-eabi, with the associated Binutils, Newlib, and GNU debugger.

5.2. Development Tools on Linux

The Linux development environment has proven to be less of a challenge in regards to the custom compiling and patching of development tools. This is mainly due to many of the tools used in this project to have originally been developed on Linux. Note that Linux was not used for the bulk of the development of DiscoBSD; portability in compiling and running of the project was the main focus.

The tools required for the MIPS-based RetroBSD are available in the documentation that comes with RetroBSD, notably, a user-compiled version of GCC targeted to mips-elf and the supporting Binutils are obtained by running the supplied build script. Supporting tools are either default programs on the operating system or are added via the particular Linux package manager. The BSD version of yacc, called byacc, is a required program that is available on OpenBSD by default but not on Linux.

All of the main and supporting tools, outlined in Section 5, that are required to compile and develop DiscoBSD are available as packages through the particular Linux's package manager. This includes GCC targeted to arm-none-eabi, and the associated Binutils, Newlib, and GNU debugger.

6. Kernel Operation Overview

Coverage of the kernel operation will be limited to the relevant issues for this project. System startup, process creation, and process management will be covered in outline in this section. For example, signals, communication facilities, and the filesystem will not be covered, but are, nonetheless, important facilities of any kernel.

The kernel gets loaded into RAM by reset and bootstrap code in the system startup sequence, and then execution is passed to it. It sets up the *swapper* process (PID 0), which the kernel will eventually become. The kernel then hand-crafts the first new process (PID 1) which will be the *init* process. The *init* process is the ancestor, and parent process, of all future processes in the system. Once *init* is created by a kernel-specific form of `fork()`, then the kernel becomes the *swapper* and manages scheduling processes.

In a roundabout and convoluted way, the *init* process loads the program `/sbin/init` from the filesystem and it is set executing. The *swapper* process eventually schedules the *init* process and runs it, which runs the `/sbin/init` executable. `/sbin/init` spawns a shell to interpret the commands in `/etc/rc`, then forks a copy of itself to invoke `/libexec/getty`, which further invokes `/bin/login` to log a user on. Upon a successful login, `/bin/login` uses a call to `exec()` to overlay itself with the user's shell. The system is now in the position that general *user mode* programs can now be run by users through their shell, and they will be scheduled and executed by the kernel *swapper* process.

The kernel uses a full swap policy wherein there can only be one process running in RAM at a time, in addition to the always-present kernel *swapper* process. The processes not currently running will be swapped out to the *swap area* on the disk, which in this case is a filesystem partition on the mounted SD card. The reasoning for this policy is that the available RAM to the system is not large enough to support multiple in-core processes. This is a defining, and unavoidable, constraint of DiscoBSD.

7. System Startup

After a system hardware reset, the kernel gets loaded into RAM from Flash by initial reset code and execution begins at the kernel's entry point, which eventually arrives at the kernel's `main()` function. Machine dependent (MD) peripherals are set up and initialized. The kernel's various data structures and services are initialized. The filesystem is mounted and set up. The *init* process is created and forked. The kernel process becomes the *swapper* to schedule all system processes. The code for `/sbin/init` is loaded from the filesystem into user memory and the *init* process "returns" to location zero of the code in user memory to execute it. The specifics of how all this happens is covered in the following subsections.

7.1. Bootstrapping and Linker Script

The default bootloader in STM32F4xx microcontrollers is set by the `BOOT0` (held low by default) and `BOOT1` (held high by default) pins. This selects the main Flash memory as the boot space, starting at address `0x00000000`.

There are two linker scripts that concern this operating system: one for the kernel and one for user executables. The former will be discussed in this section.

A linker script is a specifically formatted file that instructs the linker — as the last step of the compilation process — on how to lay out the various sections of the executable. This amounts to placing kernel code in the read-only *.text* section, initialized data in the read and write *.data* section, and specifying where the *.bss* section is located for uninitialized data and variables. The stack pointer is also placed accordingly, normally at the end of RAM for the full-decending stack on the Arm Cortex-M4. The stack pointer is defined by the label *_estack* and it is located at the end of RAM at address `0x20020000`.

A trimmed down version of the kernel's linker script is as follows:

```
MEMORY {
    FLASH (r x)  : ORIGIN = 0x08000000, LENGTH = 1024K
    RAM   (rwx)  : ORIGIN = 0x20000188, LENGTH = 32K - 0x188
    U0AREA(rw!x) : ORIGIN = 0x20008000, LENGTH = 3K
    UAREA (rw!x) : ORIGIN = 0x20008C00, LENGTH = 3K
}

/* Higher addresses of the user mode stacks. */
u0    = ORIGIN(U0AREA);
u     = ORIGIN(UAREA);
u_end = ORIGIN(UAREA) + LENGTH(UAREA);

_estack = 0x20020000;

ENTRY(Reset_Handler)

SECTIONS {
    .text : {
        KEEP(*(.isr_vector))
        *(.text*)
        *(.rodata*)
    } > FLASH
    _etext = .;

    .data : AT (_etext) {
        _sdata = .;
        *(.data*)
        . = ALIGN(8);
        _edata = .;
    } > RAM

    .bss : {
        . = ALIGN(8);
        _sbss = .;
        *(.bss*)
        *(COMMON)
        . = ALIGN(8);
        _ebss = .;
    } > RAM
}
```

All execution starts at `ENTRY(`*label*`)` where *label* is *Reset_Handler* on DiscoBSD (historically *start*). In Arm Cortex-M4, the first 32 bits (first word) of the executable is actually the address of the stack pointer, and the second word is the address of *label*. This is handled by the linker. *label* refers to a label in the architecture-specific assembly language startup code. This code will be covered in the next section.

7.2. Assembly Language Startup

The assembly language startup code differs greatly between MIPS and Arm. The MIPS startup code is entirely contained in the file `/sys/pic32/startup.S`, whereas Arm and STM has standardized on an elaborate set of files that are common amongst each family of microcontrollers. These standardized files are available from STMicroelectronics, the microcontroller vendor for STM32F407xx devices.

The following files are required by Arm for CMSIS functions:

- `cmsis_gcc.h`
- `core_cm4.h`
- `core_cmFunc.h`
- `core_cmInstr.h`
- `core_cmSimd.h`

The following files are required by STM for processor and SysTick initialization:

- `startup_stm32f407xx.s`
- `stm32_assert.h`
- `stm32f407xx.h`
- `stm32f4xx.h`
- `stm32f4xx_it.c`
- `stm32f4xx_it.h`
- `system_stm32f4xx.c`
- `system_stm32f4xx.h`

The Arm file that contains the label *Reset_Handler* is `/sys/stm32/startup_stm32f407.s` and is the file that starts all execution. This file is specific to STM32F407xx microcontrollers. Other microcontrollers in the STM32F4xx family have similar startup files, named in a comparable way.

The structure of the code in `startup_stm32f407xx.s` is as follows (shortened for brevity):

```
.global Reset_Handler

Reset_Handler:
    ldr sp, =_estack     /* Set stack pointer. */

    /* Code to copy .data segment from flash to SRAM. */

    /* Code to fill .bss segment with zeros. */

    bl SystemInit        /* Init system clock. */

    bl main              /* Call main() in kernel. */

    /* Once main() returns here as PID 1: */
    /*   enter user mode, */
    /*   run icode at address zero (to exec /sbin/init). */
    /* This is described in Section 7.4. */
```

Exception handlers and interrupt service routines are defined and handled in `stm32f4xx_it.c`. The Arm-required `SystemInit()` function, which is called from the startup assembly code shown above, is defined in `system_stm32f4xx.c`. The various header files have defines for the standard Arm environment. Once the startup assembly code calls the `main()` routine, the kernel proper is running C code and will start the kernel initialization process.

7.3. Kernel Initialization

Kernel initialization is completely contained in the file `init_main.c`, which is where the `main()` routine is located. The kernel starts in *kernel mode*.

The `startup()` routine initializes machine dependent (MD) peripherals. `startup()` is defined in `/sys/stm32/machdep.c` and is highly specific to the processor architecture and the available peripherals on the target board. For example, this is where LEDs and GPIO pins are initialized.

Kernel autoconfiguration is performed with a call to `kconfig()`, which probes for all the devices available to the system at boot time. This is a dynamic process, and as such, allows flexibility in the presence of optional devices. The absense of any required standard device will cause the kernel to panic. Kernel configuration is explained in more detail in Section 8.

The system process structure (*struct proc*) for PID 0 is set up. Each process in the system has an entry in the process table in the kernel. The process table is implemented as an array of *struct proc* entries. The process structure must always remain in main memory, no matter what state the process is currently in.

The init user structure (*struct user*) is set up. The user structure is quite unique. There are two instances of the user structure: *u0* and *u*, which are declared in the linker script. *u0* is dedicated to PID 0, the *swapper* process. *u* is the user structure of the in-core active process. The user structure of any process not currently in a *runnable* state is swapped out.

Next, signals are initialized. The kernel's various data structures, tables, and protocols are initialized. Well-known inodes are set up. The kernel clock is set up. Services are attached to the kernel. Detailed coverage of these topics is beyond the scope of this report.

The root filesystem is mounted. If no root filesystem is found, the kernel will panic. The swap file on the root filesystem is opened and cleared. If no swap file is found, the kernel will also panic. Timeout driven kernel events are started. Finally, the root filesystem is set up.

The next section will continue the kernel initialization with the final task of setting up a working kernel: getting `/sbin/init` to run.

7.4. Getting to /sbin/init

Continuing on in the `main()` routine, and following the set up of the root filesystem, the *init* process is created by the kernel-specific version of `fork()` called `newproc()`. The kernel process (as the parent process) officially becomes the *swapper* to schedule all system processes by calling the `sched()` routine, which never returns. The child process of the fork is the *init* process. In the *init* process, the code for a small assembly language routine called `icode` is copied from the kernel image to the start of user memory.

The routine is effectively the same as the following program:

```
main()
{
    char *argv[2];

    argv[0] = "init";
    argv[1] = 0;
    exit(execv("/sbin/init", argv));
}
```

The last task in the `main()` routine is for the *init* process to "return" to location zero of the code in user memory and execute it. In effect, the return is from the branch to `main()` in the startup assembly code, and is a *thunk* to run the `icode` just copied out. This process has been been, rightly so, described as "somewhat enigmatic" by John Lions in his famous Commentary on UNIX 6th Edition. The call to `execv()` replaces the image of the *init* process with the userland image of `/sbin/init`, which is loaded from the mounted root filesystem. It is especially important to understand that `/sbin/init` is running in *user mode*, not in *kernel mode*, as a regular user process.

7.5. Getting to the User's Shell

As shown in the previous section, the *init* process starts up the `/sbin/init` userland program, and exits if the call to `execv()` fails. This makes the presence of `/sbin/init` vital to the system bootstrapping procedure.

`/sbin/init` forks itself and spawns a shell to interpret the commands in `/etc/rc`, which performs various tasks such as filesystem consistency checks, and starting up daemon processes like `/sbin/cron` and `/etc/update`. `/sbin/init` then forks a copy of itself for each terminal device that is marked for use in the file `/etc/ttys`. Each copy of `/sbin/init` invokes `/libexec/getty` to manage signing on to the system. `/libexec/getty` eventually reads in a user's login name from its terminal and invokes `/bin/login` to complete the login sequence. Once the user password check is complete, `/bin/login` uses an `exec()` call to overlay itself with the user's shell (normally `/bin/sh`, the standard Bourne shell).

The system is now, finally, in a state to be commanded by users in the usual way.

8. Kernel Configuration

The kernel configuration program `/tools/kconfig/kconfig` is used to configure a kernel, based on the `Config` file in the build directory, namely `/sys/stm32/f4discovery/Config`. The support files `Makefile.kconf`, `devices.kconf`, and `files.kconf` in the `/sys/stm32` directory are used in the configuration process. Cursory coverage of `kconfig` will be outlined below, while detailed information is available from the `kconfig` documentation.

The purpose of `kconfig` is to generate a `Makefile`, which is used to compile a specific kernel. `Makefile.kconf` is a template `Makefile` that has default build rules and directives, as well as anchors to attach generated build rules. The names of specific source files used to build the kernel are retrieved from the file `files.kconf` by matching both standard kernel files and optional device drivers. `devices.kconf` contains a list of block devices and their major numbers for the filesystem.

A basic kernel configuration is possible with the following `Config` configuration file:

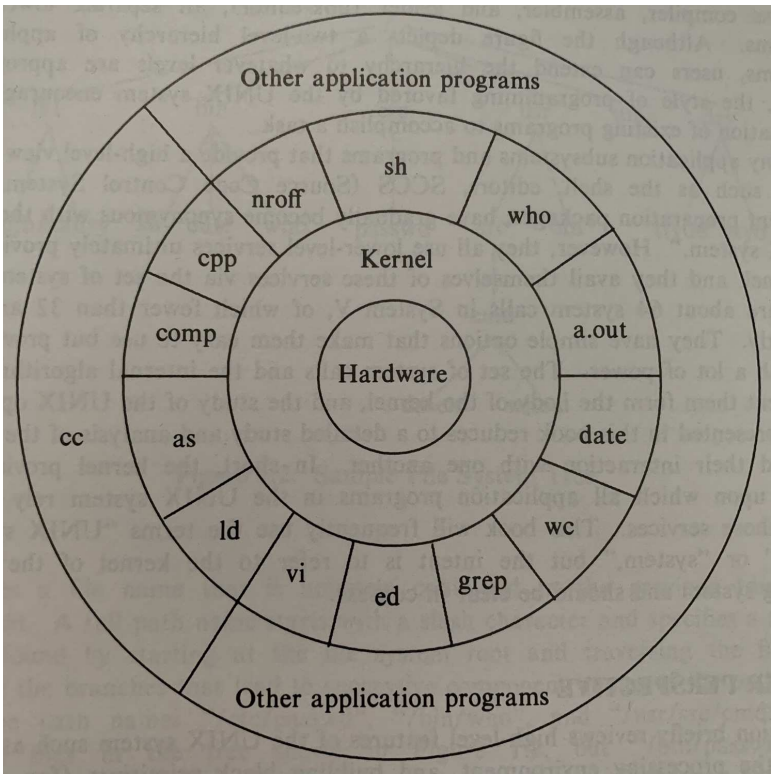
```
architecture    "stm32"                  # Processor architecture
cpu             "STM32F407xx"            # Processor variant
board           "F4DISCOVERY"            # Board type
ldscript        "f4discovery/STM32F407XG.ld" # Linker script

options         "CPU_KHZ=80000"          # CPU core osc freq
options         "BUS_KHZ=80000"          # Peripheral bus freq
options         "BUS_DIV=1"              # Bus clock divisor

config          unix    root on sd0a     # Root filesystem
                        swap on sd0b     # Swap partition

device          uart1                    # Serial UART port 1
options         "CONS_MAJOR=UART_MAJOR"  # UART1 as console
options         "CONS_MINOR=0"           # /dev/tty0

controller      spi2                     # SD card

device          sd0     at spi2 pic RC14 # SD card select pin
options         "SD_MHZ=10"              # SD card speed 10 MHz
```

Note that the full functionality of STM32-specific configuration has not yet been added to `kconfig`. A fully working `Makefile` that is able to compile the DiscoBSD kernel, using the above configuration defines, has been created by hand.

9. Userland

The userland consists of all parts of an operating system that are not part of the system kernel proper. Specifically, the shell, editors, the various system libraries, and other user programs constitute the userland. These programs interact with the kernel through *system calls*, which are well-defined entry points into the kernel that request specific kernel services, such as reading or writing to a file. This affords a separation of interests between user applications and the system and hardware management tasks of the kernel.

An image of the architecture of a Unix system is shown below:



A defining difference between Linux-based and BSD-based operating systems is that BSD-based systems are unified and complete, composed of a kernel and a userland. Linux is the kernel proper of a Linux-based system and distribution creators pair the Linux kernel with a userland of their choice, most commonly the GNU system.

The userland of DiscoBSD is not the focus of this project but it deserves cursory attention in regards to the kernel porting effort. The major areas of consideration are the C runtime startup code, low-level assembly language routines in the C library for various tasks such as string manipulations, and the linker script for the memory layout of user executables.

Userland code is completely contained in the `/src` directory. An Arm-specific directory for the C runtime startup code has been created at `/src/startup-arm` and the Arm-specific directory tree for the various C library low-level assembly language routines has been created at `/src/libc/arm`. The linker script for Arm executables is `/src/elf32-arm.ld`. The selection of Arm-specific or MIPS-specific code is dependent on specific build variables, covered in the following two sections.

10. Build System

Through the use of the previously covered host development environment, a complete RetroBSD, and mostly complete DiscoBSD, operating system can be built using the standard build features included in the codebase. The build system is structured as a collection of build variables and a heirarchy of Makefiles. The `make` build software (specifically `gmake`) manages build relations between all source files and their dependencies. The top-level Makefile orchestrates the compilation of build system tools, system libraries, userland programs and their associated manual pages, and the system kernel. The final step in the build

process is the creation of a filesystem image for installation onto an SD card. The executables, libraries, and supporting documentation are installed into the root filesystem according to the configuration in the `rootfs.manifest` filesystem manifest file. The kernel is not installed into the filesystem, rather it is installed into the flash memory of the microcontroller. This procedure is performed via the specific tools associated with the microcontroller.

The top-level Makefile has targets for all the previously outlined build steps. The standard process for building the system is to invoke `gmake` from the root directory of the codebase, which follows the creation of all dependencies until the whole system is built. Specific targets may be invoked by appending the target name after `gmake` on the command line.

10.1. Multi-Architecture Features

DiscoBSD's build system and its hierarchy of Makefiles have been amended to support the ability to host multiple architectures under one unified codebase. This development towards the concurrent support of many architectures is a major aim of this project. Two architectures are currently supported.

Compulsory environment variables is the method used to achieve support for multiple architectures. This simple method has historically been used to great success, and is exemplified in the highly portable NetBSD and OpenBSD operating systems. The compulsory environment variables `MACHINE` and `MACHINE_ARCH` choose which hardware platform and processor architecture, respectively, to use in compiling the system. `MACHINE` derives from the command "`uname -m`" while `MACHINE_ARCH` derives from the command "`uname -p`" on all Unix systems. This structure enables the possibility of future ports to other platforms and architectures.

The default platform and architecture for DiscoBSD are *stm32* and *arm*, respectively. To target the MIPS-based RetroBSD, define `MACHINE` as *pic32* and `MACHINE_ARCH` as *mips*. This can be performed either by setting the environment variables through the shell's functionality or by setting the environment variables on the command line when invoking `gmake`, as shown below:

```
$ MACHINE=pic32 MACHINE_ARCH=mips gmake
```

Another multi-architecture feature enabled in DiscoBSD, and alluded to in Section 5, is the ability of the build system to detect the host operating system and choose build and support tools that are specific to each operating system. This process is automatic and developer input is not required. The supporting Makefiles `/target.mk`, `/sys/stm32/gcc-config.mk`, and `/sys/pic32/gcc-config.mk` are responsible to selectively choose build and support tools based on present operating system features. The currently supported and tested host operating systems are Linux and OpenBSD. Although support for Mac OSX and FreeBSD was previously added to RetroBSD, these systems have not been tested, so their status is indeterminate.

11. Project Difficulties

This project has been riddled with challenges and difficulties; some small, while others were quite substantial. The naïveté of thinking that a 50 year old codebase, crafted over many tens of thousands of hours by some of the world's best computer scientists, could be fully ported with a complete kernel, in a four month semester, cannot be underestimated. The project is comprised of two overarching themes, wherein each theme had their own particular difficulties: a case study of the kernel of the RetroBSD operating system and the development of the Arm-based DiscoBSD kernel. The difficulties encountered and overcome in each of these themes will be explored in turn in the following sections.

11.1. Case Study of Kernel Operation

As RetroBSD derives from 2.11BSD, which derives directly from Sixth Edition Unix and indirectly from Seventh Edition and 32V Unix, the vast historical literature of Unix development was interrogated for knowledge of the system's operation. Although there are books written specifically for 4.4BSD, 4.3BSD, Unix System V Release 2, and Unix Sixth Edition, there are no definitive works that cover 2.11BSD. The closest is the coverage of 4.3BSD, which has part of the system ported to 2.10BSD, combined with the Commentary on Unix Sixth Edition. This made understanding the RetroBSD kernel quite challenging.

Synthesizing this disparate information, in addition to effective code tracing and single-stepping, allowed for a sufficent level of understanding of kernel operation and, in turn, enabled and initiated the code-level porting process.

11.2. Development of Arm-based Kernel

Before work on the kernel could begin, a host development environment that targets Arm processors needed to be designed and validated for efficacy. As detailed in Section 5, many different tools were required for this development effort. Of note, the QEMU-based Arm Cortex-M emulator was a challenge to port to the development environment. In addition, the older MIPS-based GCC compiler proved to be of considerable difficulty to port, and required custom patches to GCC to enable passing floating-point options between the compiler and the assembler. The modern version of the Arm-targeted GCC compiler exposed many bugs and non-critical compiler warnings. These issues needed to be fixed before development on the kernel could begin. All this work has been a part of the additional goals of the project to adapt the large-scale codebase of RetroBSD to modern and sustainable development standards.

Arm uses a standardized set of initialization and configuration files across all microcontrollers with a Cortex-M processor core. Integrating these files into the codebase of DiscoBSD proved challenging, in particular, in finding a compromise between the imposed structure of the Arm files and the historically validated structure of DiscoBSD. More work in this area may reveal an optimal and symbiotic solution.

12. Future Work

Even with the work completed throughout this project, there remains a few significant barriers to a full Arm-based DiscoBSD kernel. First, a user/kernel syscall API needs to be devised and validated. The Procedure Call Standard for the Arm Architecture could be a starting point for this work. Userland and the C library code for user executables needs to be completed and validated for proper function. This would be a long-term task, as there are potentially many difficulties that may emerge throughout this work.

Kernel drivers for useful peripherals need to be written. A UART driver and an SPI-based SD card driver would be enough for the system to stand on its own and be commanded by a user. These could be based off the embedded systems drivers supplied by STMicroelectronics. A GPIO driver would enable more fuctionality in the system with minimal implementation effort.

The implementation of kernel memory protection from user processes is a potential long-term goal. The STM32F4xx family of microcontrollers is endowed with a memory protection unit that is dedicated to this function. A system that offers reliable service must guarantee some sort of memory protection.

13. Conclusion

XXX Words

XXX Words

References

1. Serge Vakulenko, *Homepage of RetroBSD*, http://retrobsd.org (Last updated August 28, 2015). Accessed November 28, 2020.
2. Serge Vakulenko, *Homepage of Serge Vakulenko*, http://vak.ru (Last updated April 12, 2017). Accessed November 29, 2020.
3. The PDP Unix Preservation Society, Warren Toomey, (ed), *Details of the PUPS Archive*, https://minnie.tuhs.org/PUPS/archive_details.html (Last updated February 21, 1996). Accessed November 28, 2020.
4. Steven M. Schultz, *Announcement of Second Distribution of Berkeley PDP-11 Software for UNIX Release 2.11*, https://www.krsaborio.net/bsd/research/1991/0314.htm (January 1991). Accessed November 28, 2020.
5. Steven M. Schultz, *Patch level 469 for 2.11BSD Distribution*, https://www.tuhs.org/Archive/Distributions/UCB/2.11BSD/Patches/469 (April 14, 2020). Accessed November 28, 2020.
6. RetroBSD, *RetroBSD Autobuilder Revision 506*, http://retrobsd.org/wiki/autobuild.php?rev=506 (February 17, 2019). Accessed October 31, 2020.