Table of Contents

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 a simulator 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.

30 November 2020

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.

2. Relevant History of 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 Editition 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. 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

3.2. STM32 Development Board

4. Simulators and Emulators

4.1. PIC32 VirtualMIPS Simulator

4.2. QEMU-based Arm Cortex-M Emulator

5. Host Development Environment

5.1. Development Tools on OpenBSD

5.2. Development Tools on Linux

6. Kernel Operation Overview

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

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 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. The kernel's various data structures are initialized. Machine dependent (MD) peripherals are set up and initialized. Machine independent (MI) peripherals are set up and 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
}

ENTRY(Reset_Handler)

_estack = 0x20020000;

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

7.3. Kernel Initialization

7.4. Kernel Configuration

7.5. Getting to main()

7.6. Getting to /sbin/init

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

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

7.7. Getting to the User’s Shell

8. Userland

9. Build System

9.1. Multi-Architecture Features

10. Project Difficulties

11. Future Work

12. Conclusion

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.