AArch64 (ARM 64-bit) based Unix-like Operating System for the Raspberry Pi 3b

Sam Whitehead

August 14, 2021

Abstract

Blah blah blah ...

Lorem ipsum dolor sit amet, consectetur adipiscing elit ...

Contents

1 Introduction

1.1 What is ARM64?

1.2 What is an Operating System?

1.3 Why the Raspberry Pi?

2 Getting Started

2.1 Setting up the build system

For this project, we cannot use a traditional compilation setup. Normally, a program runs in a hosted environment, which means that it can use a standard library for some basic functions. In the case of C code, this is `libc`. Our Operating System will be running on bare metal, so we cannot use any of the standard header files provided by `libc`, which means that the entire C standard library is unavailable. We can still use some provided header files, including `stddef.h` and `stdint.h`, as these are provided as part of the compiler. There are other header files available in this "freestanding" environment, but these two are the main ones for the first stages of development, since they give us types such as `uint32_t` and `size_t`.

In addition to the compiler flags needed for freestanding development, we also may need a completely different compiler. If we develop our Operating System on any hardware other than a Raspberry Pi, there will likely be compatibility issues with the code generated from a regular compiler. This relates to the **Architecture** of the system running the compiler. The Raspberry Pi we are developing for has an architecture of `Aarch64`, or 64-bit ARM, whereas our PCs are likely to be `x86` or `x86_64` architecture computers. This means we need a compiler that generates machine code for `Aarch64` based systems. We would call such a compiler a **Cross-compiler** if it runs on a different architecture compared with the machine code it produces.

2.1.1 Cross-compilation terminology and tools

- **Build**: The architecture of the computer we are using to compile the program.
- **Host**: The architecture of the computer our program will run on.
- **Target**: The architecture our program builds for (only used when the program is or contains a compiler).

For this project we will be using the LLVM suite of tools, as it is designed to be a cross-compiler first, so all of the tools (compiler, assembler, binary manipulation tools, etc.) are capable of handling our required Aarch64 host system. The compiler, "clang", takes a command-line argument which indicates the architecture of the host system (LLVM ignores the difference between host and target, calling them both "target").

2.1.2 Our basic Makefile

2.2 How we will debug the OS

2.2.1 GDB

2.2.2 Serial Port

2.3 Developing on hardware

2.3.1 kernel8.img

2.3.2 Reading from the serial port on GPIO using MiniCOM on our build system

3 The Kernel

3.1 The early boot process

3.2 Memory allocation and management

3.3 The filesystem

3.4 System calls

3.5 Process handling

3.6 The init process

4 The Shell

5 Multiprocessing on the BCM2837

The processor on the Raspberry Pi we are developing for, the Broadcom BCM2837, has 4 CPU cores, so we should be able to run 4 different computations on one at the same time. This section will discuss the new challenges we can encounter as we try to get this working.

5.1 Starting the CPUs in order to run processes on them

5.2 Scheduling processes in a multicore world