Project Report

Christopher Lang

March 30, 2023

Summary

This report accompanies the 'Raspberry Pi Operating System From Scratch Guide' (PIFS Guide). It is intended to explain the purpose, motivation and technical decisions of the guide.

Contents

1 Introduction

The PIFS Guide is a complete set of instructions for building an operating system 'from scratch' - meaning that the user will manually configure, (possibly compile) and install each program into a drive they formatted and partitioned themselves. The operating system is intended for 'Raspberry Pi' [1] hardware,

[1] 'Raspberry Pi' is a brand of single board computers.

specifically the 'Raspberry Pi Zero 2 W'.

2 Motivation

Bundled operating systems, graphical installers and pre-configured operating system 'distributions' have made it easier than ever to install an operating system. However, this convenience is at the cost of the users understanding of their system. The PIFS guide teaches how to build and install an operating system from scratch, so that the user will better understand how operating systems work.

3 Intended Audience

In order to keep the guide concise and focused, some prerequisite knowledge is assumed [2]. The guide is therefore written for the user who is already familiar with how to use a Linux based operating system, but is curious as to what parts the system consists of, how each part interacts and how to install these parts independently from a pre-existing distribution.

The 'suckless' developers believe that 'In contrast with the usual proprietary software world or many mainstream open source projects that focus more on average and non-technical end users, we think that experienced users are mostly ignored'[3]. After spending much time researching for the PIFS guide, I have come to the realisation that the neglect for experience users extends to online software documentation and learning resources [4]. For this reason, I have committed to building the guide for experienced users who meet the prerequisites defined in the guide.

4 Building a Cross Compiler

Constructing all the programs in a cross-toolchain is no easy task as many of them depend on each other resulting in a 'chicken and egg problem'. For example, GCC (the compiler) can not be built before Glibc (the standard c library) but Glibc must be built with GCC! Breaking this dependency loop can be done by building the toolchain in multiple passes, each successive pass adding

[2] PIFS Guide Section 1.2.

[3] *Suckless Philosophy.* '`https://suckless.org/philosophy/`'.

[4] One particular frustration I had was when searching 'HOW does X work?' - where X is a piece of software or technical concept. Almost all results that claimed to explain 'HOW X works' actually only explained 'WHAT is X' or 'HOW to use X'. Some will attempt to provide a very basic overview of 'HOW X works' but rarely delve into any technical detail or refer to resources which do. Try searching 'How does the GRUB bootloader work'. On google.com, the top results are 'GRUB and the x86 Boot Process', 'An introduction to GRUB2 configuration' and 'What is GRUB Bootloader and What Does it Do?'.

more functionality[5]. This is a complex process and is tangential to my guide's main goal of teaching how an operating system works.

The 'Cross Linux From Scratch' guide teaches how to build a cross-toolchain[6]. However, this process consumes a significant portion of the guide - making it less focused and take more time to follow.

I have decided to download pre-built toolohain binaries from 'toolchains.bootlin.com'[7]. This will make my guide easier to follow and to maintain.

5 Build Isolation

Usually, when software is compiled, the build system will adapt the package based on the features of the host operating system[8]. However, when compiling software for a different (in this case new) system, precaution must be taken to compile using the new system's libraries. This is known as build isolation.

As the guide compiles software to a foreign target architecture, it must make use of a cross-compiler. The Linux From Scratch project demonstrates an approach to build isolation where cross-compilation is faked to prevent the build system from adapting the package based on features of the host[9]. This suggests that using a cross compiler is sufficient for build isolation but is it necessary?

Some level of build isolation is required for creating binary package archives as the binaries must run on a system other than the host. Cross-compilation can not be used because the target-triplet is the same on the host and target. I posted a question onto the 'Unix & Linux' section of Stack Exchange, enquiring about this topic[10]. As was explained in the responses to my question, the level of build isolation required for building binary packages is lesser than that required to build a new operating system. This response seems to be correct as the 'Creating Packages' page of the arch wiki[11] has no mention of build isolation - suggesting it is not a significant issue in this context. Specifying dependencies and following a file hierarchy standard is sufficient for build isolation of binary package.

A GNU 'GCC help' email thread discusses the possibility of full build isolation without a cross-compiler[12]. This correspondence came to the conclusion that cross-compilation is the only rational way to achieve full build isolation.

[5] *crosstool-NG Documentation.* '`https://crosstool-ng.github.io/docs/`'.

[6] *Cross-Compiled Linux From Scratch, Version 3.0.0-SYSTEMD-x86.* '`http://www.clfs.org/view/CLFS-3.0.0-SYSTEMD/x86_64/`'. 2014.

[7] *toolchains.bootlin.com.* '`https://toolchains.bootlin.com/`'.

[8] *GNU Autoconf.* '`https://www.gnu.org/software/autoconf/`'.

[9] *Linux From Scratch 11.1.* '`https://www.linuxfromscratch.org/lfs/view/11.1/`'. 2022.

[10] *Stack Exchange - Unix & Linux - How are binary packages isolated from host system?* '`https://unix.stackexchange.com/questions/723006/how-are-binary-packages-isolated-from-host-system`'.

[11] *Arch Linux Wiki - Creating Packages.* '`https://wiki.archlinux.org/title/Creating_packages`'.

[12] *GCC Help - host-isolated gcc without faking cross-compiling.* '`https://inbox.sourceware.org/gcc-help/fd96fa42-450a-db81-c128-9b0d03058c58@falsifiable.net/T/`'. 2021.

This is likely to be true, if there were another reliable method it probably would have been mentioned by the LFS project[13].

In conclusion, cross compilation is sufficient and (for all practical purposes) necessary for absolute build isolation. So for this project, build isolation should not be an issue.

6 Package Management and Installing Software

Installing cross compiled basic system software is conventionally done with a `make install` command[14]. This autoconf generated Makefile command will automatically copy the required files from the build directory into the final directory in the new system. This is convenient as the user does not need to worry about what files must be installed and where they should be located. However, because the files are automatically copied into an already populated directory tree, it is difficult to know what files each package installs. This can make it hard to understand what each package does. This is why the guide will run `make install` to install into an empty staging root directory. The user can inspect what files were installed into the staging root, modify them if desired, then copy them into their final directories in the new system. This way the package can choose what files to install and where to put them and the user can see what each package installs before the files are copied into a common filesystem. This solution was partially inspired by symlink style package management[15] where packages are installed into separate directories and linked into a common filesystem.

After the user has compiled basic system software, how can they install and manage additional software packages on the new system? Not all packages support installing into a non-root directory, making the staging root method difficult. Symlink style package management[16] suffers the same problem. Many modern Linux distributions often ship with a package manager which can install and manage package archives - most notably: rpm[17] and deb[18]. However, as these utilities install files automatically, the user will not have a thorough understanding of what exists on their system. This is contradictory to the goal of the guide - to teach the user how an operating system works.

Packages could be installed by using the `make install` command provided by Autoconf[19]. However, as described earlier, using this method it can be difficult to know what files each package installs. As a result, uninstalling a specific package is not easy. This method may be viable if only a few packages are required and if no packages are uninstalled.

[13] *Linux From Scratch 11.1*. '`https://www.linuxfromscratch.org/lfs/view/11.1/`'. 2022.
[14] *GNU Autoconf*. '`https://www.gnu.org/software/autoconf/`'.
[15] *Linux From Scratch 11.1*. '`https://www.linuxfromscratch.org/lfs/view/11.1/`'. 2022.
[16] *Linux From Scratch 11.1*. '`https://www.linuxfromscratch.org/lfs/view/11.1/`'. 2022.
[17] *RPM Package Manager*. '`https://rpm.org`'.
[18] *Debian Package*. '`https://wiki.debian.org/deb`'.
[19] *GNU Autoconf*. '`https://www.gnu.org/software/autoconf/`'.

Nix[20] builds each package into a unique, immutable directory without any side-effects that modify the rest of the system. Then, a 'profile' is used to symbolically link package binaries to common `lib` and `bin` directories. This approach has a number of advantages: multiple versions can be installed simultaneously, packages can be upgraded or installed without fear of breaking the system and the clear separation between package files can make the responsibilities of each packages clearer to the user. However, most package's build systems are not built with this unusual package management technique in mind[21]. As a result, it can be difficult to install packages without causing side-effects. In addition, the per-package install directories may confuse users who are accustomed to traditional Linux file system's where binaries from all packages are placed in the same directory.

I have decided to install additional software using `make install` because the guide only installs a few packages. If the user wishes to use a different method, they may, but it is not the responsibility of the guide.

7 Software Used in the OS

This section explains the purpose of all software included in the guides operating system. The reason for selecting each piece of software over alternatives is also discussed.

Raspberry Pi Firmware. The official Raspberry Pi Firmware Github repository contains 'pre-compiled binaries of the current Raspberry Pi kernel and modules, userspace libraries, and bootloader/GPU firmware.'[22]. Using this regularly updated firmware is important to ensure the latest critical bug fixes are included[23]. However, I decided not to use the repository's user space libraries because I consider the process of building ones own libraries to be a critical step in learning what an operating system consists of.

The GNU toolchain. GNU's close relationship with Linux[24] makes it the natural choice of toolchain. Its use In Linux systems is near unerversial so becoming familiar with it will be benifficial to the user. However, its complexity can make it difficult to understand. GCC was released in 1987 and has since accumulated over 14 million lines of code [25]. This makes problems very difficult to diagnose. For example, when I tried to compile GCC 12.1.0, `make` failed and reported that a number of `__LIBGCC_SF_*` macros were undeclared. Due to the size and complexity of the codebase, I was unable to resolve the issue on my

[20] *Nix.* '`https://nixos.org/`'.
[21] *Nix Reference Manual.* '`https://nixos.org/manual/nix/stable/`'.
[22] *Raspberry Pi Firmware.* '`https://github.com/raspberrypi/firmware`'.
[23] *Raspberry Pi Docs.* '`https://www.raspberrypi.com/documentation/computers/raspberry-pi.html`'.
[24] *Linux and the GNU System.* '`https://www.gnu.org/gnu/linux-and-gnu.en.html`'.
[25] Found by running `tokei` in GCC's 12.2.0 source tree after downloading prerequisites.

own so I posted a question on Stack Overflow[26]. I did not receive any replies to this question, which suggested that knowledge of GCC's complex codebase is not widespread. I was able to work around the issue by upgrading to GCC 12.2.0, but if GCC's source code was more simple, I may have been able to understand the cause for the error and solve the problem properly. In addition, GCC is partially written in C++[27], meaning that it can not be compiled on a machine without a C++ compiler. This means I am unable to natively compile GCC on the guides operating system because, for the sake of simplicity, I have decided not to include a C++ compiler.

Other C toolchains do exist[28][29][30][31]. However, many programs are written with non POSIX compliant, Glibc specific features. Support for these features in smaller standard C libraries is limited[32]. In addition, the smaller C compilers are known to produce less optimised code[33].

Custom Init System. The guide includes the source code for a simple init system written especially for the guide. By reading the code, the user can understand exactly what the init system does and therefore will have a conceptually complete overview of what is invoked in user-space. This would be more difficult with a larger, more complex init system.

Busybox. As far as the guide is concerned, what core-utils are used is irrelevant as it does not have any significant effect on the overall structure of the operating system. Busybox was selected as it is easy and fast to compile from source. This reduces unnecessary time and complexity of the guide.

Make. 'Make' is a simple build automation tool that does not have any software dependencies not already satisfied by the guide's system. With this tool, the user can install software build with the common combination of Make and C.

8 Build Process

This section outlines and justifies the guide's build process.

[26] *Stack Overflow - Failed to cross compile GCC.* '`https://stackoverflow.com/questions/73582427/failed-to-cross-compile-gcc-libgcc-sf-undefined`'.
[27] *GCC's move to Cpp.* '`https://lwn.net/Articles/542457/`'.
[28] *musl libc.* '`https://musl.libc.org/`'.
[29] *uClibc.* '`https://uclibc.org/`'.
[30] *Simple C Compiler.* '`https://www.simple-cc.org/`'.
[31] *Tiny C Compiler.* '`https://bellard.org/tcc/`'.
[32] *musl libc faq.* '`https://www.musl-libc.org/faq.html`'.
[33] *A Performance-Based Comparison of C/C++ Compilers.* '`https://colfaxresearch.com/download/7129/`'.

8.1 Preparing for the Build

A new user ('pifs') is created on the host to ensure a clean build environment [34]. Cross-toolchain binaries are installed into a directory in the 'pifs' user's home directory [35]. Environment variables are created and set to the full path of the cross-toolchain binaries [36]. Now, cross-toolchain tools can be executed by using the appropriate environment variable at the beginning of a shell command: `$CC --version`. Installing and running the cross-toolchain binaries in this way ensures they do not overwrite or are mistaken for binaries that happen to already exist in the user's host system's global path such as `/usr/bin/arm-buildroot-linux-gnueabihf-gcc`.

The guide does not require the use of any particular partitioning tool. The user can use whatever tool they are comfortable with. A 256M FAT32 boot partition and an ext4 root partition filling the remaining disk space is suggested. This allocated more than enough space for the boot files and uses a modern but sturdy root filesystem. No swap partition is used. Advanced partition tables are tangential to the guides main goal of teaching a user how an operating system works.

Files from the official Raspberry Pi's firmware repository's boot directory[37] are copied into the boot partition. `cmdline.txt` is added to pass appropriate command line options to the kernel [38].

8.2 Building Core Software

Each software is generally installed by the following procedure:

1. Extract the archive.
2. Configure the source.
3. Compile the source.
4. Install the package into a staging root: `/home/pifs/staging-root`.
5. Copy important files from the staging root into the new root file system.

The guide explains the compilation options used for each package. The staging root allows the user to inspect what files the package is trying to install before copying them into the final directory. This helps the user understand what part of the operating system each package is responsible for.

[34] PIFS guide - Section 2.1
[35] PIFS guide - Section 2.2
[36] PIFS guide - Section 2.1.5
[37] *Raspberry Pi Firmware*. '`https://github.com/raspberrypi/firmware`'.
[38] PIFS guide - Section 2.4

References

A Performance-Based Comparison of C/C++ Compilers. 'https://colfaxresearch.com/download/7129/'.

Arch Linux Wiki - Creating Packages. 'https://wiki.archlinux.org/title/Creating_packages'.

Cross-Compiled Linux From Scratch, Version 3.0.0-SYSTEMD-x86. 'http://www.clfs.org/view/CLFS-3.0.0-SYSTEMD/x86_64/'. 2014.

crosstool-NG Documentation. 'https://crosstool-ng.github.io/docs/'.

Debian Package. 'https://wiki.debian.org/deb'.

GCC Help - host-isolated gcc without faking cross-compiling. 'https://inbox.sourceware.org/gcc-help/fd96fa42-450a-db81-c128-9b0d03058c58@falsifiable.net/T/'. 2021.

GCC's move to Cpp. 'https://lwn.net/Articles/542457/'.

GNU Autoconf. 'https://www.gnu.org/software/autoconf/'.

Linux and the GNU System. 'https://www.gnu.org/gnu/linux-and-gnu.en.html'.

Linux From Scratch 11.1. 'https://www.linuxfromscratch.org/lfs/view/11.1/'. 2022.

musl libc. 'https://musl.libc.org/'.

musl libc faq. 'https://www.musl-libc.org/faq.html'.

Nix. 'https://nixos.org/'.

Nix Reference Manual. 'https://nixos.org/manual/nix/stable/'.

Raspberry Pi Docs. 'https://www.raspberrypi.com/documentation/computers/raspberry-pi.html'.

Raspberry Pi Firmware. 'https://github.com/raspberrypi/firmware'.

RPM Package Manager. 'https://rpm.org'.

Simple C Compiler. 'https://www.simple-cc.org/'.

Stack Exchange - Unix & Linux - How are binary packages isolated from host system? 'https://unix.stackexchange.com/questions/723006/how-are-binary-packages-isolated-from-host-system'.

Stack Overflow - Failed to cross compile GCC. 'https://stackoverflow.com/questions/73582427/failed-to-cross-compile-gcc-libgcc-sf-undefined'.

Suckless Philosophy. 'https://suckless.org/philosophy/'.

Tiny C Compiler. 'https://bellard.org/tcc/'.

toolchains.bootlin.com. 'https://toolchains.bootlin.com/'.

uClibc. 'https://uclibc.org/'.