Package Manager: The Core of a GNU/Linux Distribution

by

Andrey Falko

A Thesis submitted to the Faculty

in partial fulfillment

of the requirements for the

BACHELOR OF ARTS

Accepted

_____

Paul Shields, Thesis Adviser

_____

Allen Altman, Second Reader

_____

Christopher K. Callanan, Third Reader

_____

Mary B. Marcy, Provost and Vice President

Simon's Rock College

Great Barrington, Massachusetts

2007

Abstract

Package Manager: The Core of a GNU/Linux Distribution

by

Andrey Falko

As GNU/Linux operating systems have been growing in popularity, their size has also been growing. To deal with this growth people created special programs to organize the software available for GNU/Linux users. These programs are called package managers. There exists a very wide variety of package managers, all offering their own benefits and deficiencies.

This thesis explores all of the major aspects of package management in the GNU/Linux world. It covers what it is like to work without package managers, primitive package management techniques, and modern package management schemes. The thesis presents the creation of a package manager called Vestigium. The creation of Vestigium is an attempt to automate the handling of file collisions between packages, provide a seamless interface for installing both binary packages and packages built from source, and to allow per package optimization capabilities. Some of the features Vestigium is built to have are lacking in current package managers. No current package manager contains all the features which Vestigium is built to have. Additionally, the thesis explains the problems that developers face in maintaining their respective package managers.

Acknowledgments

I thank my thesis committee members, Paul Shields, Chris Callanan, and Allen Altman for being patient with my error-ridden drafts. Thank you for the feedback, the encouragement, and being diligent in responding to my questions.

I thank my most excellent friend, Rigels. You have encouraged me throughout this whole process. Every time I stop and think about how much work I need to complete, I say to myself, "Rigels is working really hard to become a dentist. I need to work hard to learn computers inside out". I will always cherish how you would greet me and ask, "What's this? Where are the commits [to the thesis project subversioning repository]? I want to see more commits". I thank you for being so interested in my project and sometimes forcing me to bounce complicated ideas off of you. I also thank you for coming to my residence one day and helping me debug the most difficult pieces of code of Vestigium. I would not have enjoyed writing this thesis without your presence.

I thank Daniel Villarreal for sacrificing his time to proofread the thesis to help me prepare it for its final draft. I thank Ryan Hoffman for showing interest in the project and for sacrificing his time to proofread the third chapter. I thank Evan Didier for showing interest in my thesis and pointing out errors in the final draft. I thank Fedor Labounko, Mike Haskel, and my uncle Leo for showing interest in my thesis project and allowing me to bounce some ideas off of you all. I further thank Fedor for answering many of my questions about the thesis formatting and writing process.

I thank the open source software developers who volunteer their free time to develop software that I use free of charge and almost free of problems. Your work inspires millions of people, including me.

Finally, I thank my father for instilling in me the discipline, will, and patience necessary for completing projects of this magnitude. I thank you for providing the resources to make everything possible. I also thank you both for showing interest in my work and for helping me overcome all of the hardships that I had to overcome in life.

Contents

0.1 Introduction

Contrast a small village with a large city. In the village, everyone knows everyone else, what everyone does, the conflicts between people, who is friends with whom, etc. In a city, the opposite is the case. One does not know the names of every person one passes. No one knows whether two individuals might argue if they are in close proximity. The village people can manage themselves without needing a large bureaucratic order, while the city remains standing only because of such a bureaucratic order.

GNU/Linux distributions are like a city. A GNU/Linux distribution is made up of thousands of pieces of software. It is nearly impossible to have a usable distribution with less than three hundred packages. To manage all of the pieces in a distribution, some sort of structure is needed. This structure is centered around the idea of package management. The purpose of package management is to allow the tracking of all the pieces of software that comprise a distribution. In this way, civil order can be maintained and conflicts avoided.

Most GNU/Linux distributions differ from proprietary operating systems, such as Microsoft's Windows and Apple's Mac OS, by how much control a user can have. GNU/Linux operating systems, depending on how their package management systems are organized, allow the user greater flexibility in shaping the operating system. This flexibility can address significant needs and preferences – more significant than, for instance, merely being able to change the background picture on a desktop. The needs in this context might include things like bug fixes, security upgrades, and hardware constraints; while preferences might involve, say, a user's like or dislike for having a minimal versus maximal set software, or their desire to optimize the system for particular hardware.

This thesis examines package management in GNU/Linux distributions and explores almost every aspect of package management. We examine the intricacies of optimizing software as well as bootstrapping compilers. We probe and describe just about all theoretical and practical ways to manage software on a system. We go as far as creating a package manager and outlining how distributions can manage their personnel.

The first chapter delves deeply into how a GNU/Linux system is optimized. This chapter introduces the reader to most aspects of optimization that can be done on GNU/Linux systems, including optimizing software code with compilers, choosing file systems, and choosing kernels. A large part of the chapter explains benchmarking techniques and presents benchmarks of optimized software and code.

The second chapter allows the reader to learn about the inside of a GNU/Linux system and appreciate the amount of time and effort a package manager saves. This chapter explains the details of a number of package managers and package management techniques. Furthermore, it describes the problems that developers face when maintaining a package manager, and provides the reader with an overview of package management from both the developers' and users' perspectives.

The third chapter explains the requirements to create a package manager. It describes the process from the preliminary theoretical stages to the practical design of code. Documentation for the implementation of a package manager forms a large portion of the chapter. Its purpose is to show, on a practical level, the strengths and weaknesses of current package management systems.

The fourth and final chapter discusses how developers can maintain a package manager. This chapter is less technical and focuses on the big picture: how a GNU/Linux distribution fits into the real world, how it can survive, and perhaps even earn a profit.

Before we can begin, we must define what a GNU/Linux distribution is. A GNU/Linux distribution is simply an operating system. We say "GNU/Linux" instead of "Linux" or "GNU" by itself in order to be more precise about the fact that the distribution consists of a combination of the Linux kernel and the GNU user space.

GNU is a recursive acronym that expands to "GNU is Not Unix". The short history behind the development of GNU software is that a group of software enthusiasts came together in the early 1990s and re-wrote the core user space applications contained within

the Unix operating system. Beyond re-writing these applications, they kept their code open to the public under a new license called the GPL.[1] The main point of the GPL is that users can distribute the source of GPL licensed software in any way they like, so long as they keep the distributed package also under the GPL.

Linux is an operating system kernel. We can think of it as a software package licensed under the GPL. The presence of "Linux" in "GNU/Linux distribution" is due to the fact that the kernel is an extremely vital part of a GNU/Linux operating system. It is what connects the software to the hardware and manages how applications take advantage of the resources provided by hardware.

We can now more rigorously define a GNU/Linux distribution as an operating system which organizes existing software licensed under the GPL license[2] and distributes it, whether for profit or not. There are over three hundred and fifty active GNU/Linux distributions.[3] Just about all of them offer some sort of distinct advantage or advantages over other distributions in terms of how they organize and bring all of the software together.

The major distributions can be distinguished by how they build what is called the toolchain. The toolchain is the most vital part in the development of a distribution. A toolchain consists of the gcc package which in turn consists of, but is not limited to, the C/C++ compiler; the glibc package, which contains the libraries for the C/C++ languages; and the binutils package, which is used by gcc for dynamic linking. The three packages of the toolchain are often patched for bug fixes and enhancements by distribution developers. Along with adjustments to the default build options, the collection of these packages makes toolchains unique to a particular distribution.

Many distributions can also be distinguished by how they patch and build the kernel package. It is often difficult to package an efficient kernel that will work on a wide range of

[1] General Public License.

[2] In practice it is difficult for many distributions to meet this definition since there are other open source licenses, as well as distributions that package non-open source licensed software.

[3] http://distrowatch.com shows statistics on distributions. It currently compares a total of three hundred and fifty nine. There are currently one hundred and ninety three distributions pending to be added to DistroWatch.

hardware. Developers of distributions have to package the kernel so that it will load only the modules needed to make all the hardware of a given machine available to the operating system. Loading support for every possible hardware configuration wastes memory.

Most distribution developers manually write the scripts, which are small, high-level programs needed in order for the operating system to initialize after the kernel loads. These scripts are called "initialization scripts" or simply "init scripts". Different init scripts often do the same things, but differ in how the user can control and manage them. Since the user can always manually edit the scripts, they are always under his control. However, the user must first learn some shell scripting to do this. Many distributions provide users with tools to control which scripts load and how. These tools are usually in the form of documented configuration files, additional command line scripts, or GUI[4] programs. In general, differences between the init scripts have to do with configuration file syntax, since few distributions are able to create GUIs or command line scripts for configuring init scripts.[5]

A novice GNU/Linux user will usually value distributions for their installation procedure. Many distributions create state-of-the-art GUIs which users can use to install a full scale distribution (with more office productivity and media applications than they will ever use) in under twenty minutes. Distributions focused on more advanced users have menu-based installation procedures, which often allow a user to have more configuration power. Some distributions offer only a guide with the general shape of the commands that a user must run and which files the user must edit. The general trend seems to be that the more advanced the GUI, the less configurable the installation. Some distributions do not even install themselves onto a user's hard disks, but run right off of a bootable CD-ROM or DVD.

Last, but certainly not least, distributions distinguish themselves by how they manage software packages. There are more than eleven thousand packages that are available to a

[4]Graphical User Interface.

[5]It is not that developers do not have the skill to create such tools, but rather that they have other priorities.

GNU/Linux operating system.[6] This large pool of packages can be useful to just about any type of user, from the video editor to a large corporation. The majority of these packages are available only by download off the Internet. Furthermore, just about every one of these packages is free. To keep track of all of these packages, distributions create package managers, which, depending on their sophistication, allow the user to search for packages, install them, update them, and uninstall them with one or a series of commands.

We have gone over the major components that make up a distribution. Now let us observe the major ways in which distributions differ. Perhaps the greatest difference among distributions consists in their respective orientation toward particular user types. For example, a distribution might focus on creating a distribution that can be maintained and used by a novice. Other distributions do not care if you are a novice or an expert, so long as you are interested in, for instance, maximum security.

Depending of the type of user that a distribution is oriented toward, distributions will have correspondingly different installation methods, package management systems, init script configuration, and documentation. Distributions oriented toward a novice will usually have GUIs for all of these components, while distributions oriented toward experts will usually have a lot of documentation in the form of "How-tos". Package managers might support a larger or smaller number of packages. Those oriented toward the advanced user expect the user to create packages for the package manager if the package is not packaged by the distribution. Some distributions' package managers might even include the ability to install paid-for, proprietary software. Some, novice-oriented distributions even provide functionality and look and feel similar to Microsoft's Windows or Apple's Mac OS.[7]

As with just about any sufficiently complicated piece of software, GNU/Linux distributions are not all perfect. They often suffer from a number of inefficiencies. One of the major inefficiencies has to do with the optimization of software packages. When a dis-

[6]This figure is based on the number of packages currently in Gentoo Linux's package management system, which can be found at http://gentoo-portage.com/Newest.

[7]Linspire: http://www.linspire.com/ is an example of such a distribution.

tribution creates a binary package,[8] it compiles it with a set of optimizations. In general, optimizations are flags that are set before compile time and used by the compiler to compile the package's source code in such a way that the resulting assembly code will execute faster on a particular processor. For example, a Pentium IV processor has newer, more efficient functions than a Pentium III, although the Pentium III has all of the functions of a Pentium IV. We can therefore compile something for a Pentium III, and it will work on a Pentium IV. However, if we do so, we will not be taking advantage of all the capabilities of the Pentium IV, and thus not making the program run as fast as it can. If we compile something for the Pentium IV, though, we will be faced with the problem that our binary will not work on the Pentium III. Distributions try to make their binaries as compatible as possible, so that they can be used on older systems. As a result, the binaries often do not take advantage of the newer, improved functions of later processors. Some distributions compile for newer processors, but as a result do not work on older machines. Some distributions do not even create binaries, but instead provide users with ways of compiling packages from source, for themselves, and for their own processor.

To make matters even more complicated, there also exist compiler optimizations that apply to any processor. For example, the gcc compiler allows the setting of the flag: `-funroll-all-loops`. This flag takes all loops in the source code and unrolls (in assembly) as many as it can. Doing this gets rid of many branch instructions in the assembly code, theoretically improving a processor's pipeline. This optimization is not processor dependent because it can be done on just about any processor used today. Whether it improves performance on a particular processor is another question. Many distributions do not like to do too many compile-time optimizations due to the fear of uncovering bugs in programs that would not have appeared, had the optimizations not been used. Because time does not generally allow the stress testing of applications using variations of compile-time optimizations, distributions tend to be conservative at the expense of building packages that

[8] A pre-compiled package, which a user can extract and use without compiling.

run at less than maximum speed. In general, distributions tend to gravitate toward decisions that will potentially prevent bugs, rather than increase speed.

Another big problem facing distributions is how to keep a user's systems stable while at the same time supplying up-to-date versions of software. Updating software in GNU/Linux is very vital. First of all, with every release of a software package, bugs are fixed and new features are added. Secondly, a distribution does not want to ignore any security holes that may be fixed in a newer version of software. The problem arises that software cannot be *too* new. If it is too new, it is untested, and thus may contain new bugs, may have breakage with other applications, or have changes that affect other applications.

The other major problem is making distributions easy to use. Even if a distribution claims to be developed for an advanced user base, it still should not make things unbearably complicated; it has to document how features are to be used. Advanced users usually find editing configuration files faster and better than doing the same thing in a GUI. Most of the time they do not wish to make their lives complicated, but rather easier.[9] Furthermore, distributions need to keep their configuration style as consistent as possible or else find a good way to communicate configuration style changes to users. Distributions for novice users need to create an easy way for users to fix things from the GUI. Distributions often have to decide between configurability and saving the users from themselves. Users with GUIs, for instance, often get the confidence to mess with things that they are not supposed to mess with, and then ask for support to bring things back to normal.

[9] Due to the freedom to customize, for example.

Chapter 1

Optimization

Optimization is a word derived from the Latin word “optimum”, meaning “the best”. Thus the act of optimizing something is making it be the best that it can be. For our purposes, we view the best as the fastest. An operating system that handles a wide variety of tasks fastest is one that is optimized. But in our eagerness to optimize, we have to keep in mind that stability is a form of optimization. Stability is the measure of how consistent something is. An operating system that locks up from time to time is not stable. Furthermore, if an operating system boots one day and not another, it is not stable.

There are a number of choices that a developer of a GNU/Linux distribution can make regarding how to optimize the distribution. Stability optimizations usually come from policies enforced by the package management system. Speed optimizations usually come from a developer’s experience with configuring a system with a number of various patches, options, and packages. We will first look at the options that a GNU/Linux developer is faced with regarding speed optimization.

1.1 Linux

The first form of speed optimizations can come from the Linux kernel. There are currently three main branches of the Linux kernel. To be more precise, there is one Linux

kernel package, and two fairly large patch sets that are designed to patch the main Linux kernel. The main Linux kernel is referred to as the `vanilla-sources`. It is labeled by kernel developers, most notably by Linus Torvalds,[1] as the stable branch. With every version of `vanilla-sources`, extra care is taken to test all of the code changes for proper functionality. Most new features are not committed to `vanilla-sources`, but to Andrew Morton's experimental `mm-sources`. The `mm-sources` are released as a collection of patches called "a patch set". Patches are files that contain a listing of lines to add to or drop from files in source code. Due to the more conservative nature of `vanilla-sources`, `mm-sources` are usually faster in benchmarks and observation. The `vanilla-sources` developers usually end up committing `mm-sources`' patches, but only after sufficient testing has been completed on each patch. The objective of the `mm-sources` branch is to provide a testing ground for considerably risky changes. There also exists the `ck-sources`, which are maintained by Con Kolivas.[2] These sources are considered experimental, but contain a different variation of patches than `mm-sources`. Some consider `ck-sources` to be a stable patch set. The `ck-sources` are mostly known for having a different CPU scheduler, called "staircase". At the time of this writing, Con Kolivas has introduced a scheduler called RSDL, which he touts as solving some of the biggest problems with CPU schedulers.[3]

We are now aware that three kernel branches exist. The question now becomes, what do these branches offer over one another? The answer to this question changes with time rather quickly. A month before this was written, `ck-sources` offered a different processor scheduler than the other two. Its scheduler, staircase, has proven to be rather stable and impressive in benchmarks. The developers of `vanilla-sources` and `mm-sources` used a scheduler developed in the `mm-sources` branch. Both schedulers are designed to work well under what are called interactive workloads. If the scheduler is a simple round-robin

[1] The creator of the Linux kernel and current lead developer. See http://kernel.org for more information.

[2] Con Kolivas is not directly affiliated with the vanilla-sources or mm-sources developers. In fact, he is a practicing doctor who allegedly learned how to program in C by reading Linux code, according to an interview by Kernel Trap: http://kerneltrap.org/node/465.

[3] One can find archives of his correspondences at the Linux Kernel Mailing List archives: http://lkml.org/.

implementation, then the user, when attempting to switch to another task, will have to wait for whatever is in queue to finish before being able to switch to the task. In other words, interactivity is the measure of how fast a user can jump from one task to another and back.

At the time of this writing, `mm-sources` and `ck-sources` come with a memory management feature, called swap-prefetching. This feature makes use of the cpu's idle time to copy to swap space[4] tasks stored in a machine's random access memory, thus saving the time which would have been lost by doing this only when the memory had become full. This feature helps a user when his memory gets filled by saving the time that it takes to swap something out of memory. There are countless other tweaks that are contained within `mm-sources` and `ck-sources`, but are not contained within `vanilla-sources`. Sometimes `ck-sources` or `mm-sources` can become slower than `vanilla-sources`.[5] A situation recently occurred where there were changes in a file related to the disk scheduler, which caused `ck-sources`' staircase to fail whenever disk load became very high. The system would lose a lot of interactivity under high disk load. As a result, a user would have been better off using `vanilla-sources`.

Some of the more popular GNU/Linux distributions compile their own kernel patch sets, with their own custom optimizations and features. Distributions that have a small number of developers tend to use the `vanilla-sources`. Most distributions, however, have methods of optimizing any kernel, regardless of the patch set. These methods are designed to reduce the amount of memory a kernel consumes.

The Linux kernel consists of parts responsible for providing various functions to the overall kernel. For example, we can configure a kernel to lack the support for swap. Likewise, we can configure it to contain support for a hardware device. Many kernel parts can be disabled or enabled. The kernel is big. If we enable all of its parts and boot into our system, the kernel will probably fill up more than fifty megabytes, a tremendous waste of

[4]Hard drive space allocated to store currently unused tasks in memory.

[5]We are considering the situation where the base version of the kernels is the same, i.e. if vanilla is at 2.6.12, mm provides patches against 2.6.12.

resources, because normally a machine only needs roughly ten percent of all the kernel has to offer. For example, we can enable support for network drivers. The kernel contains drivers for over one hundred distinct network cards. Normally, a machine has no more than two network cards and it would be a waste to build the kernel to support any more cards than the ones the machine has. The ideal kernel is one that has support for all of the functions that the user will use, but that does not use up any more memory than is required for the functionality of the functions.

Distribution developers are faced with the problem of making their distribution usable on a wide range of hardware, while preventing the kernel from being larger than it has to be for an individual user. There are two good solutions to the problem at hand. The first is to utilize the kernel's ability to configure features as modules. A module is something that is not part of the kernel that boots, but is loaded to the kernel after the kernel has booted. Currently there exists an application called `udev`, which is able to detect a user's hardware and load the necessary modules for it.[6] There is one drawback with the modular approach however: the kernel must come with a basic set of drivers for a system's hard drive bus in addition to support for file systems. That means that the kernel will have to be built with dozens of drivers for ATA and SATA disk controllers, when it will only need one or two of them. In addition, it will need support for all half-dozen file systems instead of just one. Once such a kernel boots, all of these unusable parts can be unloaded. The amount of useless parts is nevertheless significantly less than there would be if everything were built into the kernel.

The second solution is to build everything as a module, but boot the kernel with the help of what is called an "initrd image". This image contains support for all the needed disk controller drivers, file system support, and system commands. The initrd image is loaded only for the time required for the kernel to load all of its needed modules. The strategy called "bootstrapping" is simular to the purpose of initrd. Once the initrd loads, it

[6]There is more to what udev does, but it is not relevant to our discussion.

mounts the root file system, loads the modules needed by the kernel, delicately pivots its root to the system root, and unloads everything not in use. The end result is a kernel that can boot with the help of the initrd image into a wide variety of systems and take up no more memory than it needs.

1.1.1 File Systems

Most distributions offer their users a chance during the installation procedure to choose between file systems. Some distributions, however, do not provide users an easy way to choose. There are six major file systems available in the Linux kernel. The first one, considered the most tested and fail safe, is called ext2. The second one, ext3, is a journalized version of ext2, considered and shown in benchmarks to be somewhat faster than ext2. A third one, ext4, came out very recently. It is in experimental stages of development and is being designed to overcome some major limitations of ext2 and ext3. The fourth one is reiserfs. It is also a journalized file system, but shown to exceed all other file systems with its performance when working with small files. The fifth is XFS (also a journalized file system), which has proven to work well with large files. Finally, there is JFS, which shows excellent balanced performance. JFS is not better than XFS at working with big files and not as good at working with small files as reiserfs, but is good at both. Distributions that try to limit a user's file system choice often choose ext3 due to its reputation for being derived from the stable ext2 and being journalized like the other file systems. The choice of an optimal file system depends on what type of work the user needs to do. Therefore many distributions leave that for the user to decide. Some distributions focus their attention toward users with specific work loads. These distributions might encourage their users to use a specific file system.[7]

[7]The benchmarks referred to are from the Linux Gazette January 2006 Issue. It can be viewed at http://linuxgazette.net/122/TWDT.html.

1.2 Toolchain

Although the Linux kernel contributes very significantly to the performance of a GNU/Linux operating system, there exist other contributors. These performance contributors are interconnected with the Linux kernel. The contributors are the packages that compile the code that the kernel and other tools are written in. How the code is compiled can be optimized in a number of different ways. Before we discuss these methods, we must be familiar with the main packages that allow us to compile C/C++ code. We must keep in mind that C and C++ are used to write a large portion of major GNU/Linux software.

There are three packages that comprise what is termed the toolchain. The toolchain is the set of programs required to compile software. These three packages are gcc (GNU Compiler Collection), glibc (GNU C Library), and binutils. Gcc contains compilers for the C, C++, Fortran, Objective C, Ada, and even Java programming languages. A compiler is a program that translates a given language into machine (assembly) code. Glibc is the library for the C, C++, and Objective C languages. It is developed to maintain a standard Application Programming Interface for all programs that are compiled with gcc. Furthermore, glibc serves to provide "runtime facilities" to programs which directly or indirectly use the C library in such a way that they can communicate with the kernel or "access the underlying operating system".[8] Just about every language in a GNU/Linux operating system in some way uses the C Library. Therefore any program, no matter what language it was written in, indirectly uses the C library. Binutils is the dynamic linker and assembler. It allows the binaries in the system to execute properly by linking them to the correct C Library as well as organizing the assembly code.

[8] README document in the source code for glibc-2.4.

1.2.1 Glibc

A prime example of the function of glibc is how it allows tasks to initiate threads.[9] Before version 2.4 of the Linux kernel, glibc used LinuxThreads, an implementation of the standard POSIX threading library. LinuxThreads would call the kernel's `clone()` function, which would copy the task within the task's address space. The new task would then act as the task's thread. There were many problems with the way this worked, including the lack of compliance with the POSIX standard itself. The solution to LinuxThreads came with the development of NPTL,[10] which worked in conjunction with new calls created in the kernel to create threads in a manner compliant with the POSIX standard required, but with tremendous improvements in performance. Ingo Molnar, in his NPTL design paper, wrote that "In one instance starting and stopping 100,000 threads formerly took 15 minutes; this now takes 2 seconds".[11] At the time of this writing, a number of distributions are attempting to optimize their systems by making their toolchains support NPTL. A developer carefully watching developments such as NPTL, can more quickly improve his distribution's performance once such developments become stable.[12]

1.2.2 Gcc

Gcc is a package that can be used to optimize a system in many different ways.[13] When gcc compiles a C/C++ program, it translates the code into assembly language for a chosen architecture. A person using gcc to compile something has many compile-time choices that he can set. These compile-time choices[14] determine the outcome of the assembly code. We can make the assembly code advantageous for a specific processor or set flags that might have bearing on any processor. Let us discuss some of these flags.

[9] Sub tasks.
[10] Native POSIX Threading Library.
[11] Molnar's paper can viewed at http://people.redhat.com/drepper/nptl-design.pdf.
[12] This is not always
the case, as we discuss later.
[13] And also fail to optimize.
[14] Commonly called flags.

The first option is the CHOST which is also called HOST. This is set to something like i386-pc-linux-gnu. This is called the target triplet. The first part of that option, i.e. the i386, interests us most. The 86 stands for the architecture, x86.[15] The number before the 86, 3, is the revision of the architecture. For example, an i686 will work on any processor that is Pentium II or older. This includes the Pentium III, all versions of the Pentium IV, Athlon, Athlon XP, and Athlon64. An i586 will work on a Pentium or older, including all of the processors that i686 will work on. The i486 includes other older models and the i386 includes even older models in addition to models supported by a higher revision. The target triplet sets the amount of processor related optimizations. When new versions of a processor come out, they often have new features[16] that, if taken advantage of by a compiler, will produce faster assembly code. Keep in mind, that while new processors might have new features, all features of yesterday are retained. This is why we can use i386 optimized assembly code on the latest x86 processors.

The other options are the CFLAGS and CXXFLAGS variables. CFLAGS set the optimizations used by gcc, the C compiler, while the CXXFLAGS set the optimizations for g++, the C++ compiler. It is important to understand that CFLAGS and CXXFLAGS are conventional names for variables, while the CHOST is applied during the configuration of gcc, before its compilation and installation. The way that someone would optimize assembly code for C or C++ is by adding the desired optimization flags to the command line options of g++ or gcc. So, when we refer to CFLAGS, we mean optimizations for the gcc C compiler. And when we refer to CXXFLAGS, we mean optimizations for g++, the C++ compiler. There are numerous CFLAGS and CXXFLAGS, and it is often difficult to understand the consequences of them individually and combined. There are roughly one hundred different optimizations to choose from. The gcc manual provides only brief descriptions of them.

Gcc has optimization shortcuts that developers can utilize. These shortcuts are opti-

[15] Commonly called “the PC”.

[16] Usually new assembly options.

mization flags that imply a group of optimization flags. The `-O0` shortcut means no optimization at all. `-O1` or `-O` sets a minimal group of optimizations that reduce code size and execution time without significantly disabling debugging code or increasing compilation time. `-O2` includes the optimizations of `-O1`, with additional optimizations that do not increase the code size, but nevertheless increase performance. `-O3` includes all of the optimizations of `-O2` and additionally adds optimizations for those who do not care about binary size or the loss of debugging ability. Finally `-Os` sets `-O2` and subtracts all of flags that make the binary large, i.e. `-Os` optimizes for code size rather than performance.

Some believe that `-Os` is best because the smaller the binary, the faster a machine will be able to load it into memory. Others believe that `-O3` is best and does not produce code that significantly harms load time. Yet others believe that `-O2` is best because it is a balance between size and performance.[17] The best way to decide which flags to set is to consider what users do most of the time: Do they launch application after application or do they open a program and expect to use it for an extended period of time? Even if users load a number of programs, will they be more interested in how fast these load or how crisply they can get work done with the program for two hours after load time?

Beyond the optimization groups, there are optimizations that do not belong to any of the groups, notably `-ftracer`. This optimization allows other optimization to do a better job optimizing. This flag is not part of `-O3`, but has proved to have noticeable effects in benchmark performance.[18] The flag `-fsched2-use-traces`, which is used in conjunction with `-ftracer`, increases performance very slightly. In gcc versions 4.0.0 and higher, a new flag `-floop-optimize2` has appeared, which is a new version of the `-floop-optimize` flag enabled at `-O1` or above. The second version of `-floop-optimize` has negative effects in benchmarks, illustrating its experimental nature. `-ftree-loop-linear` improves loading performance in some situations and has slightly positive effects on benchmarks. An optimization that is not worth testing is `-funroll-all-loops`. As mentioned in the

[17] Performance here is viewed as fast loading times and performance while loaded into memory.

[18] See benchmark results that appear later in this chapter.

introduction, this unrolls loops, eliminating branches in the assembly code. The gcc manual cations that this flag sometimes improves performance, sometimes does not, but always significantly increases the size of binaries. When the gcc manual notes that a flag may have regressions, it is not a good idea to optimize with it. There are flags that always improve performance, but force programs to lose precision. Such a flag is `-ffast-math`. It significantly boosts floating point calculations in just about any case, but it forces significant losses in precision, potentially forcing programs to do unexpected and improper things. In general, all flags that are not declared by the gcc manual to be performance enhancing are not worth playing around with. The flags, apart from -O*x*, are carefully chosen by gcc developers who know what they are doing. Usually, only one or two additional flags added to `-O3` are sufficient for maximum performance. Which one or two to add is something that can be ironed out by trial and error with benchmarking.

There exist other options that can be added to CFLAGS and CXXFLAGS. This group of options is architecture specific. The options that we discussed above are options that work for any machine architecture.[19] The group of flags for the x86 architecture is of most interest to us, because the architecture has a diverse choice of processors and is very popular. There are a number of x86 flags that are available. Shortcuts exist for these flags. These shortcuts are applied with the `-march=[cputype]`. If we have a Pentium II processor, we would apply `-march=pentium2`. This shortcut applies what is applied by `-march=i686` (or pentiumpro) in addition to the `-mmmx` flag. `-mmmx` allows code to compile for use with the MMX feature that was introduced with the Pentium II. Likewise, if we apply `-march=pentium4`, we will get `-march=i686` along with support for `-mmmx`, `-msse`, and `-msse2`, all features included in Pentium IV processors. New Pentium IV processors come with the SSE3 feature, which a knowledgeable person can use with the following two flags: `-march=pentium4 -msse3`. This causes problems if we wish to distribute binaries optimized for a Pentium IV because either older Pentium IVs will not work or the newer

[19] Effects potentially vary from architecture to architecture.

Pentium IVs will not be optimized.

1.2.3 Binutils

We can add yet more options when compiling. These options however, are more closely related to the linker and assembler package: Binutils. The linking options are conventionally set in the LDFLAGS variable. Not too many optimizations exist for the linker and assembler. Improvements to the assembler can be noticed in the speed with which applications load. An independent developer has been working on a feature in Binutils called Bdirect. This feature was not accepted into the official Binutils package, with the argument that what Bdirect accomplishes is best accomplished with prelinking.[20] Before we can uncover the Bdirect optimization and prelinking, we must understand how binaries connect with libraries on a GNU/Linux system.

Before 1995, GNU/Linux systems used the `a.out` format for binaries. In 1995, a switch was made to ELF binaries, which are still used today. The big advantage that ELF had over `a.out` was that `a.out` required a centralized database to keep track of which libraries the binary would use and where to find them. ELF relies on a dynamic linker to load all necessary libraries at start time. The disadvantage is that it takes the dynamic linker more time to load the libraries than it takes an `a.out` to parse its database. Libraries, however, can be located in different locations on a system. As a result, it was miserably difficult to maintain the databases in the `a.out` implementation, especially as the size of applications grew. In 2003, Jakub Jelnek developed a tool called prelink. This tool was created "to bring back some of the `a.out` advantages to the ELF binary format while retaining all of its flexibility".[21] Without getting into the details of implementation, prelink utilizes the dynamic linker to collect all of the libraries that will be required by a binary and modifies the binary so that it knows where to look. Prelinking is not done by setting LDFLAGS

[20] See the mailing list thread here: http://sourceware.org/ml/binutils/2005-10/msg00436.html.
[21] Page 2 of Jakub Jelnek's paper is here: http://people.redhat.com/jakub/prelink.pdf.

during compile-time or anything like that. The prelinking tool is run on a system that is already built and needs to be re-run every time a change occurs. If we prelink the system and later install program *x*, we will need to run the prelinking tool over *x*'s binaries. Prelinking has been shown in countless benchmarks[22] to provide loading time decreases of at least fifty percent. Prelinking sounds nice, but Michael Meeks, the developer of Bdirect for Binutils, disagrees, arguing that prelink does not work well with all binaries[23] and that prelinking must be maintained by the end user. Bdirect works in a similar way as prelink – we will not get into the technical differences. Its implementation, however, comes during the compilation of a program. As a result, the Bdirect implementation can definitively work with dlopen binaries. When we compile a package, it usually allows us to set LDFLAGS. Bdirect is implemented by the developer by adding `-Wl,-Bdirect` to the LDFLAGS. Currently, Bdirect has not made it into the official binutils package, but is added by Gentoo developers as part of their patches for binutils. Bdirect is not the type of optimization that a developer would like to add to his distribution in its current state, but it is certainly something that a developer needs to keep an eye on.

1.3 Benchmarking

Now that we have discussed how a system is optimized, we can cover how to benchmark a system to ensure its optimization rather than its performance regression. It would be ideal to optimize packages on an individual basis because each package performs different sets of instructions which can be organized better under different CFLAGS.[24] However, it takes time to choose CFLAGS, compile a package, benchmark the package, test the package for bugs, modify CFLAGS, compile, benchmark, compare, modify CFLAGS, compile, etc. For practical reasons we are forced to choose a set of benchmarks to compile and

[22] Benchmarking prelink is easy because the loading times of any application can be recorded.
[23] In particular the dlopen binaries, which are becoming popular.
[24] From here on, CFLAGS additionally refer to CXXFLAGS.

run with the toolchain built for our distribution. Once we test our toolchain with various configurations of CFLAGS, we choose which flags to apply to all of the packages that will be compiled with our toolchain. With some of these programs, we might need to make exceptions and optimize in a special way.[25] If we are unlucky, the application will not work properly. The first thing to do, in the case of a problem not easily traceable, is to recompile it without optimizations, but experience suggests that such problems rarely occur.

The author has developed a simple program, written in Bash, called sysmark, that allows a developer to test a toolchain with four different benchmarks. There are six benchmarks in total: four of them test the performance of compiled C and C++ code, while the other two test the performance – particularly the interactivity – of a kernel. Two of the four toolchain benchmarks are actual benchmarks, whereas the other two are real programs whose speed depends on the performance of a processor as well as efficient assembly code for the processor. Sysmark's intended use for setting CFLAGS allows sysmark to compile and run all or a combination of user chosen benchmarks and to save or print the results. The user can then set another combination of CFLAGS and compare the results to the previous ones. It is imperative to describe each benchmark and discuss some of their results under different circumstances.

1.3.1 Scimark

The first benchmark is Scimark. This benchmark was initially written in Java and later rewritten in C. It is compiled with the gcc compiler and can help determine optimal CFLAGS. Scimark tests how quickly it can carry out a number of mathematical processes. It performs five distinct mathematical computations. The first is the Fast Fourier Transform (FFT). This algorithm is order $O(NlogN)$ and measures floating point arithmetic in addition to how well a compiler handles loops. This algorithm can be found in a lot of scientific and media-encoding software and therefore is potentially a measure of real application perfor-

[25] Usually these packages document which optimizations are inappropriate and appropriate.

mance. We use the term "real application performance", because benchmarks often focus on algorithms that may never be used in programs and thus are not indicative of anything a user will ever use. The second algorithm is Successive Over-relaxation (SOR). This algorithm is also a floating point arithmetic benchmark used in mathematical software to solve partial differential equations. Users normally would not encounter this algorithm. The third algorithm, Monte Carlo, is like SOR in that it is seen in mathematical software and measures floating point arithmetic. The fourth algorithm is Sparse Matrix Multiplication. This algorithm benchmarks integer arithmetic, and is found in mathematical software. The final algorithm is LU factorization. This is a good algorithm for seeing how well floats, integers, and arrays are handled, and is also found predominantly in mathematical programs. Although most of the algorithms fall outside of the "real application performance" category, they are excellent for viewing which CFLAGS might improve performance in one algorithm, and degrade performance in another. For example, on a Pentium IV 3.07 GHz machine the following results can be noticed:

CFLAGS	`-O0`	`-O3`	`-O3 -march=pentium4`	`-O3 -march=pentium4 -funroll-loops`	`-O3 -march=pentium4 -ftracer`
FFT	141.12	310.36	317.13	317.13	318.29
SOR	446.05	458.80	458.80	450.22	460.99
Monte Carlo	53.90	81.34	167.25	167.77	167.25
SparseMatMult	231.58	782.52	801.66	766.50	804.12
LU	348.30	1054.31	1110.03	1351.82	1128.37
Average	244.19	537.47	570.97	610.69	575.81

All numbers are in mega flops, the compiler version used was gcc-3.4.6 with glibc-2.3.6.

We can see from the results that the unoptimized code with `-O0` is the slowest. A tremendous boost is given with `-O3` optimization and additional boosts are provided with `-march=pentium4` and `-ftracer`. A somewhat significant average boost is given with `-funroll-loops`. Observe how the `-funroll-loops` optimization increased the results of two algorithms, Monte Carlo and LU, and degraded the performance of another two.

`-funroll-loops`, as predicted by our previous descriptions, optimized performance in certain areas while degrading it in others. This example shows that `-funroll-loops` is not a good optimization flag to build an entire system with. `-ftracer`, on the other hand, is a good optimization because, because it boosts the performance of all of the algorithms tested, even if only slightly.

1.3.2 Bashmark

The next benchmark is Bashmark. Bashmark is written in C and designed to test the performance of applications. In our terms, it tests the quality of code compiled by the toolchain packages. Bashmark consists of five relatively simple tests. The first of the tests includes doing a great deal of integer arithmetic. The second test examines floating point arithmetic. The third reads and writes to memory with varying amounts of data. The fourth part tests memory deallocation and allocation, a measure of how well programs can be loaded into or unloaded from memory, how well they collect garbage, and how well they clean up when exiting. The last part tests multi-threading, or how well process threads can be launched. Bashmark is a benchmark whose results do not represent real application performance. Applications do not purposelessly add and subtract numbers. Bashmark nevertheless helps us see how various CXXFLAGS affect results. Usually we keep CFLAGS and CXXFLAGS the same to have a certain level of consistency. On the Pentium IV machine the following results were recorded:

CXXFLAGS	`-O0`	`-O3`	`-O3` `-march=pentium4`	`-O3` `-march=pentium4` `-funroll-loops`	`-O3` `-march=pentium4` `-ftracer`
Integer	1109	1209	1603	1601	1512
Floating Point	21	23	23	23	23
Memory r/w	609	1183	2805	2809	2810
Memory de-/alloc	290	506	505	505	505
Multithreading	728	698	701	702	703

The numbers are bashmark's "score" values. The toolchain is the same as with the previously shown scimark results.

First of all, the floating point test shows very poor results. There are two possible reasons for this. The first is that the Pentium IV does not handle floating point arithmetic well, and the second is that the compiler does not produce good floating point code for the Pentium IV. The results could also be due to a combination of the two. The results show several unexpected drop offs. For example, the unoptimized code scores 728 in the multi threading test, while the rest show at least 25 points less than that. The multi threading benchmark is not always consistent on a symmetric multiprocessing capable processor such as the Pentium IV that the benchmark had been run on. It probably has something to do with daemons and other applications that run on the machine and occasionally give up the extra thread slot for a number of threads. Bashmark is not the most consistent benchmark out there, but one can really see the differences with optimized and unoptimized code. We can see slightly more solid results on an Athlon64 1.8 GHz processor using gcc version 4.1.1 and glibc 2.4:

CXXFLAGS	`-O0`	`-O3`	`-O3 -march=athlon64`	`-O3 -march=athlon64 -funroll-loops`	`-O3 -march=athlon64 -ftracer`
Integer	627	1164	1171	1177	1185
Floating Point	172	1149	1148	1148	1154
Memory r/w	1693	1757	1694	1717	1700
Memory de-/alloc	571	701	589	655	662
Multithreading	2339	2430	2415	2357	2345

The numbers are bashmark's "score" values.

The lack of a drop off with `-ftracer` on the integer test perhaps indicates that on a Pentium IV the `-ftracer` flag should not be used for CXXFLAGS. We do not see much of a harmful effect with `-funroll-loops`, but also do not see any benefit either. Also notable is the lost performance when setting the `-march=athlon64` flag. Theoretically, we would expect that flag to give us a nice lift – especially in floating point arithmetic – because it takes advantage of Athlon64's SSE3. However, we saw slightly degraded results. We have to keep in mind that gcc version 4.1.1 is still a fresh development branch with a

large number of regression bugs requiring fixing. If we look at results completed by a toolchain with gcc version 3.4.6, on the same Athlon64 machine, we can see significant advantages in memory read and write as well as integer tests. We also see significant drops in floating point and memory deallocation and allocation tests. The CFLAGS used under 3.4.6 with the benchmarks were `-O3 -march=athlon64 -ftracer`. The bashmark benchmark forces us to really think about what might be going on in all parts of the system. It is best to take bashmark's results with a grain of salt.

1.3.3 Lame

The Lame part (no pun intended) of the sysmark benchmark suite takes a real program and notes the time that it takes to complete an operation. Lame is a wave to mp3 encoding program. A wave file is a recording of uncompressed audio, while an mp3 file is encoded.[26] First, a five hundred megabyte wave file is generated, then Lame is instructed to encode that file. The file is generated from a section of one of the author's old essays for a social science course. The wave file was generated with Festival, a program with the ability to convert text into audio. The small wave file is copied and assembled into a five hundred megabyte file. Lame – intelligent program that it is – normally notices that the large file has many repetitions and encodes only what is not repeated. To force Lame to encode the full five hundred megabytes, it is passed the `-r` option. Lame is written in C and compiled with gcc. Lame produced the following results on the Pentium IV:

CFLAGS	`-O0`	`-O3`	`-O3 -march=pentium4`	`-O3 -march=pentium4 -funroll-loops`	`-O3 -march=pentium4 -ftracer`
Time Taken	8 : 59	4 : 35	4 : 25	4 : 31	4 : 25

Values are in minutes:seconds.

[26] An encoded file can be thought of as compressed.

Lame produced expected results. Notice the regression with `-funroll-loops`. This time `-ftracer` did not improve or degrade anything. This shows that `-ftracer` can give us boosts or not change anything at all.

1.3.4 Povray

The next benchmark, Povray, a program written in both C and C++, is a persistence-of-vision ray tracer. Povray is used for video editing, in particular, creating high quality three-dimensional graphics. Video editing is a very intensive operation, which involves countless different algorithms and manipulation of vast amounts of data. Povray comes with a benchmark settings file along with a collection of scenes to do encoding on. Povray takes anywhere from thirty minutes to thirty-five minutes on the Athlon64 on which we conducted benchmarks. As much as we would like to have a comparison of how Povray responds to various CFLAGS, such data is impractical to gather due to the length of the benchmark, especially on `-O0`. The benchmark was conducted on a number of distributions:

Distro	Arch	Crux	Debian	Fedora5	Gentoo	LFS	Slackware	Trustix
Toolchain	gcc-4.0.3	gcc-4.0.3	gcc-4.0.3	gcc-4.1.1	gcc-4.1.1	gcc-4.0.3	gcc-3.4.6	gcc-3.4.4
	glibc-2.3.6	glibc-2.3.6	glibc-2.3.6	glibc-2.4	glibc-2.4	glibc-2.3.6	glibc-2.3.6	glibc-2.3.5
Total Time	31 : 37	32 : 07	31 : 43	31 : 38	32 : 59	31 : 37	33 : 47	33 : 56

The following CFLAGS were identical in all tests: `-O3 -march=athlon64 -pipe -ftracer`. Values are in minutes:seconds.

The results yielded by Povray are somewhat surprising. First of all, note that Fedora Core 5 and Gentoo have identical toolchains, but Gentoo lags in the results. Fedora Core 5 is an i386 optimized system and theoretically should lag significantly behind Gentoo, which is i686.[27] Povray is an extremely consistent benchmark,[28] so the margin of error is a matter of seconds at most. The best explanation for this anomaly is the type of patches

[27]The Gentoo Linux is the author's optimized distribution of choice.

[28]The author has run it over on selected systems and it has always yielded results within a one second differential.

the Fedora and Gentoo developers might have added to the toolchain packages. Fedora and Gentoo are known to add patches that speed up their toolchains. Note also the results that came out of two i686 optimized distributions running identical toolchains: Arch and Crux. The discrepancy in their results is also very odd. Crux does not add too many special patches – just a few patches that fix bugs. Arch does not add any patches to gcc whatsoever, but adds a number of them to glibc. We can also see that when Povray was compiled by older compilers, its performance dropped accordingly.

1.3.5 Kernel Benchmarks

The next two benchmarks are kernel benchmarks. Interbench serves to measure kernel interactivity, which, as we have mentioned earlier, is how fast a system responds to a user's task switching. Lmbench stress tests a GNU/Linux system and outputs detailed results concerning all sorts of kernel functions. Benchmarking with these two benchmarks would have yielded interesting results half a year ago, before `vanilla-sources` adapted a large number of patches that had set `mm-sources` and `ck-sources` apart in their performance. Currently, not much of a noticeable gap exists between the kernels. We must utilize these benchmarks when new patches are adopted into the branches to see where the advantage lies. The developers of `vanilla-sources` are very wary of introducing new, untested code into their branch, even if the code induces significant improvements. As a result, the experimental branches occasionally come out significantly on top in performance.

Chapter 2

Package Managers: Making Life Easy

Though many novice users do not agree, package managers make life with GNU/Linux very easy. Their argument goes: why can't I just install what I want with the program's installer? The problem is that very few packages actually have the type of graphical installer that these users presume. The vast majority of packages are installed by means of the command line interface.[1] Developers of the packages release their software either as a source package that needs to be compiled or as a package for some sort of package manager. If a user wishes to avoid the package manager, most of the time he will have to know his away around the CLI to unpack the source, run the configure script or scripts, run the `make` scripts, and then clean up if desired. If a user does not know which scripts to look for and run, then he will usually look for documentation provided in the software package. Most of the time the documentation gives the basic steps for installing a package. However, if the user needs to do other steps or configure a package to his liking, he will have to read a lot more documentation and sometimes even the source code itself. Although not always the case, a package manager is often useful, because it allows a user to get what he wants without reading any extra documentation.

Another tremendous advantage of a package manager is the ability to uninstall packages. When a user installs a program from source, half of the time there will not be an

[1] CLI.

uninstall script. Furthermore, any generated uninstall script might be deleted when a user cleans up the package's temporary building and extracting directory. A package manager is usually designed to track which files a package owns and allows users to uninstall the package with a single command. Before we discuss why a package manager is such a special piece of software, we will first describe a brutal Linux From Scratch installation procedure. This procedure shows the main motivation behind the creation of package managers. It also reveals what a package manager does behind the scenes.

2.1 Linux From Scratch

Linux From Scratch, LFS, is considered by many to be a GNU/Linux distribution. LFS does not give its users any media to install from or any packages to download, with the exception of initialization scripts and patches. LFS gives its users instructions in the form of a book to be downloaded, purchased, or read online. This book contains a full command by command installation guide. A user who follows the book bootstraps a compiler, builds a functional toolchain, and then uses it to build vital GNU/Linux system packages. Upon completion, a user is free to follow the Beyond Linux From Scratch book, where he can follow further instructions on how to install packages that he might want to use on a day-to-day basis.[2]

The LFS procedure is time consuming, even if the user sets all variables in such a way that he can get away with copy and pasting commands. If a user wishes to deviate even a little bit from the LFS book, he might end up consuming yet more time.

LFS starts out by setting up a safe building environment. This safe environment consists of the creation of a new user, partitioning space on a hard drive, mounting it, setting a variable describing the location of the mount point, customizing the user's system variables such as PATH, and fixing permissions in such a way that the user can only write where we

[2]Like Firefox, OpenOffice, and other applications that require a large number of packages to be installed on the system as dependencies.

want him to. LFS then proceeds first to create a separate, minimal toolchain, which will be used to build a complete toolchain. Building something small at first to build something large is referred to as bootstrapping. LFS does a bootstrap of a system because, a user's host system might have a compiler unable to build properly a fully functional toolchain. It can build a minimal toolchain, but problems might arise in building a full toolchain. A user using ICC[3] has no assurance that ICC will compile gcc properly. Bootstrapping assures that the resulting compiler and tools will not be broken.

Creating a temporary system[4] begins with the installation of binutils. The temporary system is installed in a special directory and, because binutils is the linker, we install it first so that everything else links to libraries in the special directory instead of to the host system's libraries. When installing binutils – just like most other packages – we first download its source code.[5] We extract the source code as the user created for the LFS install. We create a build directory outside of where we extracted binutils and run binutils' configure script from inside the directory where we plan to build binutils. Had LFS not provided options to use to the configure script, we would have to read binutils' documentation. Configure scripts, conventional in GNU/Linux programs, first check whether your system has the needed tools to compile the package, then generate other scripts, used in later steps to build and install the program. These generated scripts might vary depending on what the configure script discovered and also according to what options were passed to the configure script. These options range from choosing what parts to leave in or out of the program to specifying the location where the program will install.

If one is fortunate, one can pass the `--help` option to see what they can change in the configure script. If the developers of the package updated what the `--help` option displays, then the user will see all of the options. If not, then the user will have to search for the developer's documentation. Also, since the output of `--help` is condensed, it is very difficult

[3]Intel's C Compiler.

[4]LFS calls this the minimal system.

[5]LFS provides links for where to download source code. Had LFS not provided this, we would have to either use a search engine like Google or know that we can find many of the packages on gnu.org.

for an unknowledgable user to understand what the options actually do. Therefore such a user will have to find the documentation relating to the various options. An inexperienced GNU/Linux user might spend hours searching for and reading documentation just to install one relatively small package such as binutils. In a way, LFS reads all of the documentation for a user and highlights the important parts. When building binutils for a temporary system, LFS instructs users to apply two options: `--prefix=/tools` and `--disable-nls`. The first option tells the script to install the program into the `/tools` directory, the "special" directory mentioned earlier. The second option tells the script to generate scripts that do not install NLS, the native language support feature. The NLS feature allows binutils to work for users who desire to build a system that works in a language other than American English. Because this is not a permanent binutils, LFS attempts to trim down binutils as much as possible. Trimming down packages during the installation of the temporary system is consistent with the spirit of bootstrapping and debugging.

After everything goes well with the configure script (we will later discuss what happens when it fails), we build the package using the `make` utility. By convention, the configure script generates scripts called make files, special scripts run by the `make` utility. Why not use an ordinary script and not have to rely on `make`? The reason lies in the fact that the `make` utility is built "to determine automatically which pieces of a large program need to be recompiled and issue the commands to recompile them".[6] In other words, when `make` fails[7] and the user makes an adjustment and then resumes `make`, `make` will resume from where it left off, unless the user tells `make` to do otherwise.

The `make` utility also makes it convenient for developers to create options for a user to choose from. Configure scripts are not `make` scripts because the configure script needs to run from beginning to end in order to collect the information needed to generate `make` files. It is important to keep in mind that configure scripts are written to be compatible with a multitude of shells and systems that might not use the `make` utility. This means

[6] GNU Make Manual Page.

[7] To be more precise, this is when the command that make invoked failed.

that if the programmer modifies it accordingly, a configure script can generate scripts other than `make` files.[8] After the configure script runs, by convention, running a simple `make` builds the package, that is, compiles it. If a developer did not follow conventions, then he should have specified in the documentation what option to pass to `make` in order to run all procedures. After building the package, we then run `make` with the install option, that is, `make install`. This copies the compiled binutils package to the actual directory that we set as an option to the configure script.

Having installed binutils, we proceed to install gcc. When we install the gcc package for the temporary system, we install only the compiler for the C programming language. The reason we only need the C compiler is that all of the software installed for the temporary system is written in C and so a C compiler is sufficient to compile the compiler that will be used permanently. After running the configure script, we do not build the package with a simple `make`. We pass the bootstrap option to `make`. The bootstrapping option has nothing to do with the overall bootstrapping procedure being done. The generated `make` files in gcc build gcc by bootstrapping it. Without getting into too much detail, a tiny portion of gcc is built. This tiny portion is then used to build a larger portion and the larger portion is used to build yet another larger portion with more functions, et cetera, until we get a compiler that can compile C programs. After gcc bootstraps, we install it as usual.

Installing binutils and gcc was relatively easy, but getting glibc installed is at least twice as difficult. First of all, before doing anything with glibc we need to copy a "freeze" of kernel headers to the appropriate place. Without going into details, the Linux kernel used to put a number of header files into a system's include directory where header files are conventionally placed. The problem was that these headers would change drastically from one kernel version to the next, and an independent Linux-libc-header project was started "to maintain an Application Programming Interface stable version of the Linux headers".[9] Second, there are about eight configure options, most of which trim the resulting package down, but some

[8] This is a very rare occurrence.

[9] LFS, 5.5.1.

of which would normally require more than the usual amount of documentation reading in addition to experience. One of the options enables NPTL, described in the first chapter as something that we want these days. This option is not something like `--with-nptl`, but rather `--enable-add-ons`. Another option is `--with-binutils=/tools/bin`. This tells glibc `make` scripts to compile using the binutils located in our temporary system directories. As we shall see shortly, this not an obvious option to add. After that, the regular `make` and `make install` is run. However, this is not all. The minimal toolchain is installed, but needs adjusting. This adjusting step is where LFS users make most of their mistakes. First, LFS instructs the user to backup `ld`, an application installed by binutils, and replace it with an `ld` which links programs to programs installed in the directory of the temporary system. Then a user runs a series of commands that locate gcc's "specs" file, runs a sed command that enters the specs file, and replaces the directory of the host system with the directory of the temporary system. This allows the new toolchain to use itself, without relying on the host system's toolchain. In other words, anything compiled by the LFS user is compiled with, and only with, the toolchain just built.

LFS wisely instructs its users to see if the correct toolchain is used by telling them to create a simple C program, compile it, and read the binary file that results with `readelf`. The output of `readelf` must display that the binary will request the new binutils as the interpreter. If this is not displayed, then the user either forgot to do something, did not do something correctly, or deviated from the LFS book without correctly considering what additional things might need to be done or not done. The slightest of mistakes can be made. For example, the LFS book has its users set the PATH variable into the shell. The PATH variable tells the shell where to look for executables. If we make an executable and wish to launch it from any location without entering the location of the executable, we either place the executable in a directory contained within the PATH variable or add the directory in which the executable resides to the PATH variable. The variable looks like this: `/usr/local/bin:/usr/bin:/bin:/opt/bin`. It is a list of directories separated by

colons. The directories to the left of the colon have precedence over those to the right. This means that if two executables are both called `readelf` and are located in `/usr/bin` and `/bin`, the shell will launch the one that is in `/usr/bin`. If we place `/bin` in front of `/usr/bin`, then the shell will execute the one in `/bin`. When LFS users set the PATH variable, the first location in it is supposed to be the temporary system directory. This allows the new toolchain executables[10] to be used when they are available rather than the host system's. Using the new executables consequently allows the the user to build a temporary system linked to its temporary directories.

If the user did not make any mistakes, he then moves on to install a few packages required to test the toolchain and proper functionality of other packages. When the user finishes that, he goes on to build a bigger, stronger toolchain. The order of installing the toolchain package mattered before and continues to matter at this stage. First, gcc needs to be built, but this time with support for C and C++. Although this is still not the final compiler that will be built for the final system, it is just as robust. The new gcc will be built in such a way that it links to the glibc libraries located in the temporary system. After gcc, binutils is built with the newest gcc. Glibc does not need to be rebuilt, because it was already linked to the libraries in the temporary system. Here is where the toolchain is solid enough to be used to install a minimal set of programs that will allow the temporary system to do everything, without needing any help from the host system. These programs include the `make` utility, shell,[11] `tar` utility, `patch` utility, Perl, and more: about twenty in total.

Once the temporary system is ready, the vital chroot[12] procedure is run. The chroot procedure involves first configuring the temporary system's shell, providing it necessary variables such as PATH, mounting devices and random access memory from the host system,[13] and finally changing root from the host system's / to where the LFS partition is

[10] That are installed in the temporary system directories.

[11] Bash.

[12] Change root.

[13] This is so that the temporary system can communicate with the kernel's memory and task management correctly and access devices that it might need.

mounted and loading the new shell such that it believes that / is /mnt/lfsmountpoint – or, wherever the partition was mounted, if a partition was even used. The new shell only sees a /tools directory in which the user has been building. The shell does not see the host system. PATH is now set such that the /tools executable directory is last and before it are the conventional /bin, /usr/bin, etc. The idea is that the final system will be built to gradually replace the temporary system until it becomes an independent system capable of rebuilding itself and much more.

This is exactly what is done after the directory structure is created; among other things, glibc is installed – after Linux-headers are copied of course. This time, a number of patches are applied to glibc,[14] localization abilities are built, and a full scale glibc is installed into the shell's root, which is the final system. The toolchain adjustment procedure done before is redone backwards so that everything built links to the final system's libraries rather than what is in /tools. Binutils comes next, followed by gcc. Both are patched, built with full scale abilities, and installed upon the completion of a number of checks. If the new toolchain does not work and link correctly, the user can usually continue, but will break the system if and when he deletes /tools. If the user knows about this and starts looking for where errors were made, he might end up backtracking all the way to the beginning and basically re-doing everything that was done, which might take at least six hours to complete.

If everything works as it should, the user proceeds to install the minimal set of programs required to boot into the new system. Upon building these packages, the user might receive compile failures or, if he is more fortunate, configure scripts complaining about the lack of a library. Sometimes the user can find a workaround to get the package working correctly, but only until the workaround causes a problem much later on. For example, package foo's configure script fails because it can not find header bar. The user finds the header in the normal /include directory and guesses that foo might want the bar in its source directory

[14]These patches fix bugs and make improvements.

where the user is building the package. The workaround works and the package is built. After three hours of installing additional packages, another package fails. After hours of troubleshooting, the user determines that although `foo` found `bar` and built successfully, the second package now attempts to use `foo`, but can not do so properly because `foo` found `bar` in the wrong place. The better workaround would have been to edit `foo`'s configure script to search in the `/include` directory for `bar`, rather than in the source directory.

If one follows LFS without any deviations or mistakes it is highly unlikely that any problems will occur. Any customization usually requires a lot of time and hassle. Everything has to be done manually. If, for example, a user wishes to make his final toolchain with a horde of patches used by other distributions, the user will first have to download the patches. If he is lucky, the patches will all be in one place. After that, he might want to read them to see which ones are really needed.[15] Applying the patches will be even more annoying. The LFS book provides links to patches which are all consistent.[16] When a user applies patches originating from different hands, the user has to spend extra time guessing which strip option to use so that the patch applies. A patch contains a collection of file names to modify followed by which lines of those files to modify. The file names can differ depending on how the patch was created. The file name paths might or might not start from a package's top level build directory.[17] To apply a patch that starts with the top level directory one would have to add the `-p1` option to the patch, which would strip that leading directory. If the patch does not come with a leading directory, then we would have to use `-p0` with the patch to prevent any stripping. We could open the patch and see what stripping options to apply, but if many patches are applied manually, it takes a lot of time – not so much processor time as typing time – to open them, put together the `patch` command, and then repeat. It becomes faster to try the patch with `-p0` then `-p1`. This has

[15] Why do the extra work of applying a patch for gcc relating to flags for the MIPS architecture if the user will not be using that architecture?

[16] LFS also provides links to patches downloaded from one location: LFS's servers.

[17] If we extracted gcc version 4.1.1, by default it would be in a directory named gcc-4.1.1, its top level directory.

to be done with many patches and becomes very tedious.

When building a GNU/Linux system from scratch, a user might encounter the situation in which two distinct packages install the same files. Both packages might build two libraries for example that are installed in the same exact place, but might function slightly differently. LFS instructions tell the user when this happens and tell the user what to do in order to avoid overwriting files that might not be good to overwrite. If a user does not follow the excellent LFS instructions, then he might not notice that files are being overwritten during installation and never figure out the reason why something broke later on.

All of these things are problems that one faces when using GNU/Linux on one's own. However, these are only gripes compared to even larger problems. The first of these problems has to do with uninstalling packages. Let us say that we installed our system, booted into it, configured it to our liking, and installed many more programs, but now we want to uninstall something that we do not like. If we are very lucky, we did not delete the source directory in which we built the package. If we are even more lucky, a `make` file in the directory will contain a listing of uninstall commands. If we are not that lucky, then in the worst case scenario we will have to install the package and figure out which files it wrote to our file system. After that we will have to write a script – unless we have the patience to do it command-by-command – that deletes all of the files. Yet, even this is not the peak of the troubles we might face. What if we receive information that application `foo` version 3.4 has security bugs that are fixed in version 3.5? We will want to update `foo-3.4` to `foo-3.5`. The easiest course of action would be to just install `foo-3.5` hopefully overwriting `foo-3.4`. Doing it the easiest way might not be the best way, because there might be files that `foo-3.5` does not overwrite because `foo-3.5` does not need to use those files. We would therefore have useless files lying around on our hard disk.

Updating simple packages is not nearly as difficult as updating a toolchain package such as gcc. Newer versions of gcc have regression and bug fixes. To make use of many of these fixes, we first will overwrite the files of the old gcc with the newly built ones.

Then we will have to recompile every single package compiled with a gcc compiler.[18] To recompile every package, we will have to go through most of Chapter 6 of the LFS book, followed by reinstalling all packages that we installed afterward. Not surprisingly, this is time consuming, boring, and even more so if any mistakes require troubleshooting. A person might have to work eight hours a day just to maintain their system and fix problems with it. Documentation provided by the LFS project certainly brings this value down, but not so much so that a person would actually use such a system for serious purposes.

2.2 Package Management Techniques

Luckily, package management provides the solution to the torture brought about by an LFS system. The package management system just described is perhaps the simplest one: keep everything in your head.[19] This, as we have described, is very inefficient. We can think of working with GNU/Linux the LFS way as we would view writing programs in assembly. We can do anything with assembly, it just takes a lot of time. The same is true with an LFS system – we can do anything with it, it just takes a lot of time. To accomplish large tasks, higher level languages are created which group functions of assembly together. Likewise, package management systems are created by grouping together things done on a primitive level in an LFS system. There are high level programming languages and then higher level programming languages. Perl, for example, is a higher level language than C because it groups together much more than C does. The same applies to package managers. The highest level ones, such as portage, yum, and aptitude, group many more operations together than rpm, dpkg, and ports. All of these group even more operations than the package management techniques that we will discuss in the upcoming paragraphs. We have described the lowest level and now are ready to discuss the higher levels. On the level

[18] This is to take advantage of fixes to improve resulting assembly code.

[19] Hopefully memory will be good enough to fix problems when they arise and to remember which programs wrote which files.

just above the LFS level are a number of techniques. These techniques revolve more around policy than actually writing any helper scripts.

2.2.1 Directory and PATH Method

The first technique is installing every single package in a different directory and modifying variables such as PATH to include all of the individual package directories. This way, uninstalling packages is easy, as there is no fear that a package might overwrite files that we do not want to overwrite, and updating a package involves only a fairly simple change in the PATH variable to point to the new package. Perhaps the greatest benefit to this package management technique is easily having multiple versions of packages installed on the same system. Having multiple versions of the same package allows us to quickly revert to an older version if the new version does not work. This can be very handy if we update a program that other programs depend on. However, if the program has some ABI[20] changes, it might require that other packages be rebuilt. If we need to use those packages more than we need to work with the new package, we change our PATH variable to give precedence to the older package. Uninstalling a program is just a matter of removing references to it in system variables and deleting the package directory.

This sounds great, but there are drawbacks. A normal user will probably install around five hundred packages on the system. This means that the PATH variable will have five hundred entries. This also means that the configure options when installing packages will be very messy. What about libraries and headers that packages usually install into `/usr/lib` and `/usr/include`? We will have to somehow force packages to look for libraries and headers not in one central location, but all over the place. Therefore, maintaining such a system can become just as cumbersome as doing so without this technique. We will not only have to edit a bunch of variables, but also get our ELF binaries to search in appropriate locations – no easy matter.

[20] See section 2.7.2.

2.2.2 Directory and Link Method

Another technique also involves installing packages in separate directories, but instead of fixing up a set of system variables, linking everything to the appropriate directories. This solves any problems with ELF binaries and having to maintain tremendously long PATH variables. However, it creates other problems. First of all, once we install a program, we will have to link every single file of that program to the appropriate directories. This problem is lessened if we are adept at writing scripts that do this for us. However, we will not get away with just deleting the directory where the program is installed. If we created hard links, then deleting the directory will not delete the inodes of the files in the file system, which means that the program will not go away. If the links that we created were soft links, then what we have left are a bunch of files that point to nowhere. We can of course delete the symlinks first by automating the process with a script. However, the problem of programs overwriting files returns if we rely on automated scripts. We do not overwrite files however, but rather links, which is not so bad, but we will still have to clean up the mess by pointing a bunch of links to the right places. In short, this technique is also quite cumbersome.

2.2.3 Timestamp Method

Another technique is designed to improve the ability to update packages. This technique takes advantage of timestamps. Every file on a GNU/Linux system has three timestamps: access time, modify time, and change time. The access time is when the file was created. The modify time is the last time that a file was modified. The change time is the last time when an external change was made to the file, such as the creation of a hard link. When we install a package, we record the range of time during which it was installed.[21] If we wish to uninstall it, we can easily write a script which uses the `find` utility to find all files created

[21]The best way to do this would probably be to run a script that records the program's name and its corresponding start of installation time and then run another script that records the time when the package finishes installing.

during that range of time. We can update any package, in the same manner, by cleaning out any files installed by the old version which are no longer needed. The only thing that we have to worry about maintaining is a database of timestamp ranges for everything that we installed. Although we can not install multiple versions of the same programs with the same ease as through other techniques, we gain a tremendous advantage in simplifying how we know which files belong to which package.

The timestamp approach is not without major drawbacks, however. Users may not install different packages at the same time. This can be done accidentally. For example, on daylight savings day, the clock shifts back one hour. If the user installs a program one hour after installing another program, the timestamps for both will be the same and when updating one of them later, the second one will unexpectedly break. The timestamped approach also can not be used if there is the possibility that two or more people will log in to the machine and install packages at the same time or if one person is in the process of installing a large program, but simultaneously decides to "save" time by installing another, smaller package.

2.2.4 Users method

There is yet another package management technique. In this one, every package is installed as a separate user. What this accomplishes is that we can uninstall programs by using `find` to track down all files that are owned by a package user. The user name is the package name, so we do not have to remember any names – nor any passwords, because no one will log in as them. Installing programs in this manner provides benefits, but is somewhat involved. We not only have to create a new user every time, but we also have to set permissions such that incoming packages can write to the system without overwriting existing files. How this is done is somewhat tricky and we must be on the same page regarding our understanding of GNU/Linux permissions. Permissions in GNU/Linux are divided into three sets. There are permissions for the owner of the file, for the group to

which the file belongs, and for everyone else. Normally, software is written so that the owner of the file[22] has full privileges, the group the file belongs to has read and execute privileges while everyone else has also read and execute privileges. There is also the ability to make things "sticky". This permissions attribute allows for writing to a location without overwriting any files . In the user package management technique, package users are added to an "install"group, which allows the package user to write but not overwrite. This way, when a package is installed, it prompts when a package is bringing in files that already exist.[23] This gives the person in control of the installation the preemptive warning needed to decide what to do: either back the file up or delete it.

Updating packages is also a little bit tricky. The person updating can first delete all files owned by a package, or back them up and then install the new version of the package. This, however, presents problems for some packages. For example, the Coreutils package contains programs that allow files to be moved and copied around.[24] If we remove this package for an update, then we will not be able to install the update unless we move files out in such a way that we can temporarily modify the PATH environmental variable. Updating a package such as glibc is also a nail-biter because, you have to move out vital libraries. You first have to build the package before moving the libraries out because otherwise gcc will not have the libraries needed to build the new version of glibc. These are not really disadvantages, however. In fact they are advantages, because it allows the person to think about what is going on and act accordingly. The disadvantages come when a package is shipped to update its permissions to 4755.[25] In such a case, a person using the package users method has to edit the package's appropriate scripts: either the configure scripts or `make` files. The edit process can be cumbersome, and if it is automated there is the potential that something other than permissions will be modified. Another disadvantage is that a number of packages may wish to update `/sbin/ldconfig` in order for dynamic

[22] Most packages assume that this is root by convention.
[23] A permission denied error appears.
[24] These programs are known as mv, cp, ln, etc.
[25] Owned by root, writable, executable, readable by root, and executable and readable by everyone else.

linking to function properly. However, since the permissions have the "sticky" attribute, the packages can't write to `/sbin/ldconfig`. The packages' `make` files usually do not say anything about this, and a package ends up being broken for a reason unknown to the person behind the keyboard. The user, therefore, always needs to remember to log in to root and set `/sbin/ldconfig` to be writable by the install group.

2.2.5 LD_PRELOAD Method

Now we come to yet another package management technique. This one is called the LD_PRELOAD method. Upon the installation of a package, a library is preloaded which tracks selected calls to the file system. The calls that are tracked are those that create files. When a file is created, it is added to an installation log for the package. Once a package installation completes, a log file is left with a list of all files that were installed. We can easily do things such as uninstall packages using the installation logs. Tracking whether or not files are being overwritten is also not difficult to do. We can either use the tracking library or search through the log files for the possibility of a collision. Updating packages can be done by comparing logs after installation to determine whether or not any files were left behind by the old package.

The main drawback of LD_PRELOAD is that the libraries have to be loaded at the correct time. When, for example, a `make install` needs to be run, we have to remember to load the libraries. It would not be a good idea to load the library for a configure script, because we would track files created only for the installation process. We also have to remember to unload the library afterwards to prevent logging non-installation phases. Remembering to load the library is trivial compared to writing the library. Not only will one need to have advanced knowledge of the internals of commands such as `mv` and `cp`, one will also need to be fluent in the C programming language. Furthermore, the library – the LD_PRELOAD implementation specifically – will not be portable to systems that do not have dynamic ELF binaries.

2.2.6 Temporary Build Directory Method

The most popular technique used by distribution developers is to build a program in a temporary directory, note all of the files, and copy the files to the file system. The temporary directory is within a fake tree so that the package can be properly linked. This technique has all of the strengths of the LD_PRELOAD or timestamp approach, but requires more time and hard disk space to implement. Normally a package is built and its files are copied to the system directories. In this technique the files are first copied to the fake tree, and then to the real tree. This means that we need twice the hard drive space and have to spend twice as much time copying files a second time. If a package is large, we really feel these weaknesses, especially if we are tight on hard drive space, which is not uncommon if there are many very large packages installed on a system. The greatest strength of this technique is that in the fake root, modifications can be made to a package before installing it. This technique is the most popular because developers use it to create binary packages. Binary packages are shipped by a distribution, allowing users to get software without compiling it on their own computer and without having to track its files. The techniques we have discussed earlier are all used by either enthusiasts, people with narrow purposes, or developers of package managers.

2.3 Binary Packages

We have had a taste of how tedious it is for a user to work with LFS and some low-level package management techniques. Fortunately a user does not have to touch any part of them because of the existence of binary packages. To install a program we just extract a pre-compiled package to its default directory and proceed to use it. We do not have to spend time running configure scripts, the compile scripts, and install scripts. Huge packages like OpenOffice can take ten hours just for the compile scripts to finish. With a pre-compiled binary package we do not have to wait more than five minutes for the whole program to be

installed and be usable.

There are, however, tremendous drawbacks involved in using binary packages. The first is that the packages will not necessarily be optimized for the system to which the user is installing. Just about all distributions that ship binary packages optimize for one type of all-encompassing processor. As we have discussed in the previous chapter, binaries optimized for i386 will encompass both the earlier x86 architecture processors and the latest x86 architecture processors. Second, users might get a binary package which lacks a feature they want or includes a feature they do not. When we run configure scripts, we can make the scripts generate compile scripts which allow for the compilation of extra features. We can also turn off features to save space if we need to. A binary package might lack a needed feature or be needlessly big, forcing the user to compile the package from source or necessitating a search for a separated binary package. Third, if a new package or package version is released users have to wait for a binary package to be created and placed on a distribution's download repository. There might be more weaknesses, but these are the major ones. However, because of the advantages that binaries provide, most distributions base their package managers on being able to handle these binaries.

2.4 Primitive Package Managers

We have described the lowest level of organizing packages in GNU/Linux: no package manager. We have also described some techniques which are a level higher. We can now proceed onto the next level: actual package managers. Package managers, by themselves, can be divided into higher and lower levels. We will start with the lower level package managers and gradually rise to the higher level package managers.

2.4.1 Pkgtool

One of the most primitive package managers can be found in the Slackware distribution. This package manager is actually a collection of shell scripts that come as a package called pkgtool. When we refer to pkgtool we are talking about Slackware's package management system. As we will see, pkgtool is rather primitive compared to other package management systems. Slackware has a binary package repository. Currently, the binaries are i486 optimized and are compressed as `tgz`[26] archives. There are two common ways of compressing files and folders in the open source software world. Initially a tape archive (tar) is made from a collection of files. This `tar` does not have any compression and is simply a conglomeration of files. Tar files are then compressed with either `bunzip` or `gunzip`. The `bunzip` format can be used if there is a desire to minimize size. The disadvantage with `bunzip`, however, is that it takes longer to both compress and decompress. Archives compressed with `gunzip` end up being larger than `bunzip`, but require less time to be decompressed and compressed. Slackware developers, to judge by their choice, seem to favor giving their users the ability to install the binaries faster. Of particular interest to us are four scripts that come with pkgtool.

The first is `installpkg`. A user runs `installpkg` on a downloaded binary. The script simply extracts the files in the binary to their appropriate locations. The inside of a Slackware binary contains files starting with the root directory. If a package has a file that goes into `/usr/bin/`, then the binary package is going to have a `usr` folder with a `bin` subfolder and finally the file itself within the `bin` subfolder. Installpkg parses a package directory structure and replicates it in the real system. In other words it copies the files to the correct locations. Some packages also come with an install script in `install/doinst.sh`. The script is executed if `installpkg` finds it. All additional install scripts do any extra things that `installpkg` can not do, but the developers want to be done.

The next script is `removepkg`. When `installpkg` installed a package, it logged what

[26] tar file compressed by gunzip.

files were installed and where. Removepkg parses the log files[27] and deletes all of those files. The next script is `upgradepkg`. This script is applied by a user in the same way as `installpkg`. Upgradepkg takes the new – or old[28] – `tgz` package, installs it using `installpkg`, and then checks the logs of the old package that was installed before. If a file of the old package was not overwritten by the new package, it is deleted. In other words, anything belonging to the old package is not needed by the new package if it was not brought in by the new package. To be extra sure that nothing was removed that should not have been, `upgradepkg` does the installation of the new package a second time.

Last, but not least, there is the `makepkg` script. This script is normally not used by users. It is used by Slackware developers and those who wish to correctly create a Slackware package. What `makepkg` does is best described by the comments in its source code: makepkg "Makes a Slackware compatible '*.tgz' package containing the contents of the current and all subdirectories. If symbolic links exist, they will be removed and an installation script will be made to recreate them later. This script will be called 'install/doinst.sh'". A person using `makepkg` under normal circumstances first runs a configure script on the program for which he wants to create a binary. After that he runs the generated make scripts to compile the package. Once the program is compiled, the person figures out which files normally would be installed to the system with `make install`.[29] The person copies the files to a directory of his choosing, runs `makepkg` in that directory, and has his binary package in the same directory. A developer probably tests the package by installing it and attempting to use it. If something is wrong,[30] he will have to start all over and re-package the program.

Slackware's pkgtool is primitive because users have to install and upgrade packages one at a time.[31] Furthermore, they have to manually synchronize themselves to Slackware's package repositories to keep their packages up to date. Other inconveniences include the

[27] These are stored in the /var/log/ directory.
[28] The script works in cases where a user wishes to downgrade.
[29] He will have to see the make scripts for that information.
[30] Maybe he should have used different configure options, for example.
[31] If a user knows his way around a shell will take advantage of globbing.

lack of any dependency tracking. A user might install a package which will not work. Usually a package will complain, when launched on the CLI, about a missing library file. If the user is lucky, the library file goes by the name of a package with that library file. If he is not, then he will have to search the Internet for which package the library belongs to. We shall get to dependency tracking a little later.

2.4.2 RPM

Slackware's pkgtool is a collection of tools that creates and manages package archives. The Red Hat distribution, along with countless others, uses a considerably more refined package manager and archive format. RPM, The Red Hat Package Manager, has a long history and a ton of features. We will describe the major features and relevant history. RPM was written in 1997. Before it, the Red Hat distribution used RPP,[32] PMS,[33] and PM.[34] RPP was known for its ease of use for users when installing packages, but had the disadvantage that it was difficult to create packages for it. RPP was a source based package manager, which would need programs' source code modified to fit its needs. This made development difficult for RPP. PMS built packages from original source code and made modifications as it built them. Although PMS made it easier on developers, it lacked a number of features for querying and file verification. PM had all of the features of RPP and PMS, but turned out not to be as good a solution as RPM. RPM was initially written in Perl. The developers wanted to use a language built for being able to write code quickly. With Perl however, RPM quickly grew too big[35] for it to be used on a floppy disk, or to install a system with. The RPM developers rewrote it in C, making it both faster and smaller. RPM came to combine the ease of use of RPP and the developer friendly character of PMS. Additionally, it had some new features.

[32] Red Hat Package Processor.
[33] Package Manager System.
[34] Package Manager.
[35] It required Perl, which was quite large, during installation.

One of these features was its ability to handle configuration files. Slackware's pkgtool assumes that users will backup their configuration files before updating a package. RPM, on the other hand, checks packages' configuration files and determines whether to overwrite them, leave them alone, or warn the user while leaving the new configuration file with a file name extension next to the old configuration file that it did not overwrite. What RPM does in this regard is quite simple. When the package tries to write to the configuration file directory, `/etc`, it checks whether or not the file already exists.[36] If it does not, then it writes the incoming file to disk. If a file of the same name as the incoming file already exists, then checksums are taken of both files and an additional checksum of the original configuration file that came with the old package. If all checksums are the same, then the files are the same and the existing file is overwritten. If the original file checksum is the same as the file currently on the system, but the incoming checksum is different, then the new file overwrites the existing one. This can be done because the user never modified the configuration file and therefore the new configuration file can only take advantage of new features of the updated program. If the the checksum of the incoming file is the same as the checksum of the original file, then nothing is done because the configuration files did not change and the user modified the file to his liking. If all checksums are different, the user is warned and the original file is renamed with a `.rpmorig` extension and the new file is written to disk. It becomes the user's job to figure out what changed in the new configuration file and judge whether he might need to rewrite in his own configuration file.

Other features RPM was built with included the creation of its very own archive format. Whereas pkgtool simply used the existing `gunzip` format, RPM designed its own format to allow for access to information without extracting the entire archive. For example, if upon installation of a package pkgtool wished to display the description of the package to the user, it would have to extract the whole archive. This is costly for a large package or on a slow system. RPM on the other hand is able to display such information by extracting

[36] It additionally tracks some other directories.

only a small portion of the archive. Archives for RPM are of course in the rpm format and post-fixed with a `.rpm` extension.

While on the topic of file naming, now is a good time to explain the way RPM systematizes package names. Pkgtool and just about all other package managers have almost identical conventions, so this is more of a universal issue. A package name consists of the program name, followed by the version of the program, followed by the release of the program version by the package builder, and optionally followed by the type of architecture, optimization level, or type of package it is. The program name and version depends on the conventions used by the program developer. The release of the package is determined by distribution developers. Initially, they release a package to users without specification of a release milestone. However, they later might find a bug, and in turn release the package of the same program and version, but with a `-x` after the program version at the end. This tells both users and package managers that this is a newer package and can be applied as an upgrade to the older package. The next part of a package name is optional, but almost always appears with rpms. It follows the program version and release and might look something like `.i386`, `.src`, or `.fc5.i386`. We already know that the `ixxx` shows the optimization level that the package was compiled with. It also shows that the package will work on an x86 system. The `.src` means that the package is not a binary package, but a source package. The `fc5` is an example of a package built specifically for the Fedora Core 5 distribution.[37] If an rpm is designated as `src`, that means that it is an archive of the original program source code along with patches and instructions for how RPM will build the package.

The rpm archive contains a lot of information about a package, including statistical information regarding things such as the size of the package, dependencies required by the package, the time and date when the package was built, checksums of each file, permissions of each file, and descriptions of the package and files. All of the information is used to

[37] This is currently Red Hat's community project.

quickly query a package with RPM. The information is also used during the three main tasks that RPM is designed to accomplish: installing, updating, and removing packages. Installing an rpm package is done by passing RPM the `-i` option and feeding it the location of an rpm package. This location can be on the machine's file system or on the world wide web. A user can invoke a number of options with `-i`. These include disabling the installation of documentation, disabling RPM's ability to handle the overwriting of existing files, setting a different installation path,[38] and a number of other less important options. When RPM installs normally, it first checks for dependencies. It checks if all required dependencies are in the install database on the system. If a dependency is not there, RPM stops and tells the user that he needs to install that dependency first. Next, RPM checks if any files might get overwritten. If so, RPM stops and tells the user. RPM of course stops if the package is already installed. If all dependencies are present and there are no file conflicts, RPM runs any pre-install scripts that exist in the package. These pre-install scripts might include removing any files that might get in the way. RPM then determines which files are configuration files, and runs the checksum comparisons to determine what it will need to do. Once all of this is sorted out, RPM extracts all of the files and copies them to where they are supposed to go. RPM then runs any provided post-installation scripts.[39] Finally, RPM finishes by adding information to the database. During the entire installation procedure, RPM keeps track of the files that were installed, their checksums, permissions, and to which package they belong.

To remove a package, a user invokes the `-e` option to RPM along with a package or list of packages. RPM parses its database, determines if the removal of the package will break any other packages,[40] executes any existing pre-removal scripts, saves copies of configuration files if they had been modified, deletes all of the files belonging to the package, executes any existing post-removal scripts, and finally removes entries for the package in

[38] A different root.

[39] This may include updating /sbin/ldconfig.

[40] In other words, if the package is a dependency for them.

the database. RPM stops if the package is a dependency for any other package and tells the user. We have already discussed how RPM knows if configuration files have been modified.

The third main task RPM accomplishes is upgrading packages. To upgrade a package users do the same thing that they do when installing, with the exception that they invoke the `-U` option. RPM does the same things that the install task does with the exception of performing the configuration file checksum comparisons. After the installation of the new package is completed, RPM uninstalls the old version or versions. It does not remove any files of the new package, but only files that are owned by the old package.[41]

2.4.3 Dpkg

Although RPM does a lot of things, it is not the only package manager that does these things. Dpkg, Debian's package manager, is similar. Dpkg also works with its own archiving format. To be specific, the archiving format consists of an "ar" archive containing three files. The dpkg packages end with a `.deb` extension. Like RPM, dpkg packages contain information regarding dependencies of the package, files of the package, scripts, etc. The first file[42] of a Debian archive is the version of the archive. This is important because the archive's internal formatting changes[43] from time to time. To retain compatibility, dpkg contains C code to handle every different archiving version that ever existed. We are describing format version 2.0 of `.deb` archives.

The second file is `control.tar.gz`. This is a `gunzip` archived file which contains a series of text files with information about the package. This information includes the package name, type,[44] installed size, the developer responsible for its maintenance, version of the package with revision, dependencies, description, listing of configuration files that the package might bring in, pre-install script, postinstall script, pre-uninstall script, and

[41] This is because of the way the database was adjusted during installation.

[42] The "ar" archive is designed to open the separate files without extracting the whole package.

[43] Even the set of required information changes.

[44] This includes architecture, branch of Debian to which it belongs, and whether or not it is an essential, required, important, standard, optional, or extra package.

post uninstall script. Debian packages do not need a list of files that the package will attempt to install because it logs that on the fly during installation.[45] Dpkg, like RPM, also supports the ability to install source packages. Dpkg's source packages similarly contain an archive of the original source code, with a `.dsc` file containing information about the file. Dpkg, however, does not have a number of patches in the source archive like RPM, but a file with a `diff.gz` extension. `diff.gz` is one big patch with all modifications that a developer made. The `.dsc` also contains instructions for building the package. Dpkg will not install a built source package directly; there is no single command that a user writes to build it in the first place. Dpkg builds the package and then builds a binary `deb` archive which the user can install. A user using dpkg installs packages with commands similar to those of RPM. The installation will log the files installed, handle configuration files, run the scripts if there are any and append any other relevant information to the dpkg information directories on the system. Installing with dpkg will stop if existing files might get overwritten or if dependencies are lacking. There are two ways a user can remove a package: with the purge option or remove option. The purge option removes the package and its configuration files, while the remove option removes the package files, but not the configuration files. A user can also run a dpkg with a `--configure` option. This will run postinstall scripts and re-write all configuration files to disk. Dpkg does not have an option that users can use to upgrade. The install option handles that task automatically.

2.5 High-level Package Managers

Dpkg, RPM and pkgtool are still all low level package managers. Although they are all easy to use compared to doing everything manually, they still force users to do a number of tasks manually. These tasks include having to find the package and where it is located, finding dependencies if needed, and knowing when to and when not to apply overriding

[45] It is almost certain that some archive version might have had a requirement for that information.

options in the case of file collisions. Although these things seem acceptable prices to pay for solid file tracking and configuration file handling, imagine if we have a system with six hundred packages, all of which we want to upgrade to new versions. First, we would have to find web links to the latest versions. Furthermore we would not even know about a new version being available if we do not visit a distribution's repository. When we begin installing the new versions, some of the packages might have new dependencies.[46] We would have to find the new dependencies' packages and install them. We would be lucky if we finished everything in the course of three uninterrupted hours. Now imagine if we have more than one machine on which we wish to upgrade all of our packages. With pkgtool we can always get away with downloading all packages that are available on its mirror and "globbing" all of the files into `upgradepkg`. With RPM or dpkg, we might not be able to glob, because we might be providing it non-globbable web links to archives. A package name starting with `z` might need to be installed before one starting with `c` so that RPM or dpkg do not complain about dependencies.[47] One can understand why there are package managers that allow users to find out about new versions of packages, update all packages with one command, download packages automatically, and automatically bring in dependencies in the right order.

2.5.1 Apt

According to Debian's documentation, dpkg is designed as a low level tool for developers and users with special needs. Users are encouraged to use apt[48] to take care of any updating, installation, and removal needs.[49] Apt is not actually a package manager, but rather a frontend for dpkg, although philosophically speaking it is a package manager. We will get more into frontends later, but it is vital to examine what apt does for a user so that

[46] This will be even more of a pain to determine with pkgtool due to the lack of any dependency information.

[47] We are assuming that the package starting with c specifies a need for a dependency whose version is greater than of the currently installed version of the package starting with z.

[48] Advanced Packaging Tool.

[49] http://www.debian.org/doc/manuals/reference.

we can more easily understand what is special about the package managers we examine next.

First of all, apt has a list of all packages that are available. A user can at any time update that list by typing `apt-get update`. This list is used by apt to automatically track which packages are installed on the system and have new versions available. This allows users to save a lot of time that might otherwise be spent searching for new versions of a package. Second, apt automatically downloads packages from mirrors. Mirrors are servers that contain copies of the files contained on a main distribution server where new files are added. In our case the files are new packages. Apt automatically downloads anything that it needs. This includes the list of newest packages and the packages themselves. In other words, with apt, the user does not have to search manually for package files on the Internet. Third, apt allows the user to search for text in the names and descriptions of all packages. With dpkg, we would have to do so on a package by package basis. Furthermore, apt also contains a database of all files that can possibly be installed by all packages. The user can use apt to search for individual files. This is extremely helpful if a user is looking for an executable, but the executable is not part of a package description or name. Fourth, apt automatically queues up dependencies and does so in the right order. Whereas dpkg does not let the user install the package until the user installs all dependencies,[50] apt automatically downloads and installs all dependencies for a package along with the package. Fifth, apt allows a user to update every single package with one simple command: `apt-get upgrade`. Upgrading is synonymous with updating in our terminology. If a user has six hundred packages, he can run this command, eat lunch, come back, and apt will have completed a task that would have taken more than several hours using dpkg and probably days with LFS. Sixth, apt does not just remove packages but also removes a package's dependencies in such a way that the removal of the dependencies will not render any other programs malfunctional. In other words, apt checks which dependencies are needed by other packages and does not

[50] Don't forget that the installed dependency might itself require a dependency.

remove them, only removing those dependencies which belong to the package that the user is attempting to remove. This removal feature is also applied when the system is upgraded and when deprecated packages might need to be removed.

If you think that this is enough for a complete package management solution, think again. Apt has a feature where a user can run `auto-apt run command`, where `command` can be just about any command, but is usually a configure script for a source package.[51] If the configure script complains about the lack of something, for example the g++ compiler, apt will pause the running of the script, call apt's apt-get to install g++, and return to the configure script as if g++ had been installed in the first place.

Everything that apt does underneath is calling various options to dpkg. It gets all of its information from dpkg[52] and does almost everything using dpkg. When it installs packages, it asks dpkg if the package is installed; if it is not then it will look at its package list, determine dependencies, ask dpkg which dependencies are already installed, queue up the needed dependencies, then call dpkg with appropriate links and options to install dependencies of dependencies first, followed by dependencies of the package, and then the package. If the package already exists, then it will check the latest version available in its list and compare it to the version installed reported by dpkg. If the installed version is older, it will do the install procedure, but additionally queue up for removal any dependencies that will not be needed afterwards. Upgrading works the same way, except that apt evaluates every single package.

2.5.2 Swup

Swup[53] is the package manager used by the Trustix distribution. Trustix is a distribution oriented toward users who run servers on the Internet and who want a distribution whose

[51] http://packages.debian.org/stable/admin/auto-apt.
[52] With the exception of the auto-apt feature and things like the package list.
[53] The Secure SoftWare UPdater.

priority is security.[54] There is nothing extremely special about swup. It allows a user to do the things that apt does, with just about the same ease of use – although it can be argued that apt is easier to use. Swup is, like apt, a frontend with the exception that it works with rpm packages. Swup does not use the RPM package manager, but rather its own libraries and tools to manage rpms. Due to this, we can consider swup to be a package manager that works almost entirely from the ground up. The reason we are going over swup is because it allows us to examine what security features package managers can be built with.

Swup keeps checksums of all files and allows a user to validate files at any given time. Swup downloads a latest package list just like apt. It provides users with the assurance that they are downloading from a trusted mirror by issuing special GnuPG[55] signatures. GnuPG downloads all data in an encrypted fashion, eliminating the possibility of anyone intercepting packets and sending malicious ones to the user.[56] It assures – without getting into technical details available in GnuPG documentation – that whatever the user is receiving, is from Trustix and untainted. Swup is designed to update packages. Trustix is not a bleeding-edge distribution which distributes the latest versions of software, but rather one which makes sure that all software is secure and quickly updates software to a new version when a vulnerability is found. Swup also attempts to bring in minimal dependencies. The argument is that the fewer packages, the fewer chances of security failure.

A big feature of swup is that it enables the user to maintain easily different roots on the system. Having a separate root, to which only chroot access is possible, is an extra security layer which allows isolation of security breaches to smaller segments. Swup supports a simple option `--root`, which allows a user to update packages in a specific root, without having to chroot. Using the `--root` feature also allows extra minimal installations within the root. The way the `--root` feature works is quite simple. It simply extracts packages to the selected root, and logs what it did in that root. When updating, it reads the logs

[54] http://www.trustix.org/.
[55] GNU Privacy Guard.
[56] http://www.gnupg.org/.

contained in the root and acts accordingly. Actually, this feature is found not only is swup; RPM and dpkg also allow packages to be extracted in different roots. It is simply significant that with swup users need only add one extra option to swup. The fact that swup holds checksums for all files is also not new. Rpm does the same thing and can verify the files and rebuild those that do not match the correct checksum. Swup, because it uses rpm packages underneath can pretty much take advantage of all the security abilities provided by RPM.

2.5.3 Pacman

Swup is not a package manager which manages packages entirely by itself. It uses RPM's archiving format. Pacman,[57] however is a full scale package manager. It is developed by the people who created and maintain the Arch Linux distribution. Pacman uses the `tar.gz` archiving format, identical to Slackware's `tgz`.[58] Pacman's archives contain a data file called PKGBUILD which provides the following information: the package name, version, package description, web link to the package's website, a list of files to treat as configuration files, dependencies required for building the package from source, dependencies required for the package to run and work, a web link to the source file or files, checksums for the files, and the commands needed to build the package from source. There are also optional preinstall, postinstall, pre-remove, post-remove, pre-upgrade, and post-upgrade scripts. Finally the package contains the binary files that were built with the PKGBUILD file using `makepkg`,[59] the Arch Linux developers' tool which is used to create pacman binary archives. The `makepkg` utility builds packages from source in a similar fashion to RPM or dpkg in that it installs the built package into a temporary root. Then, `makepkg` is used to package the files in the temporary root into `tar.gz` archives to be used by pacman. Pacman is designed to calculate and install dependencies, delete unneeded dependencies,

[57] Short for package manager.
[58] The archives formats are the same; they just different file extensions.
[59] Not to be confused with pkgtool's makepkg.

track configuration files the same way that RPM does, update the entire system like apt, and provide information. Pacman is unique in giving the users the option of "freshening" a package or "upgrading" it. The upgrade process involves removing the package first and then bringing in the new package. "Freshening" does what the other package managers do when they update packages: bring the new package and then remove the left over files. Pacman uses a master package list that pacman downloads when a user supplies the `-y` option. This master package list, just as in apt allows pacman to figure out whether or not there are any new packages available and install the latest packages upon a user's request.

2.6 Centralized Package Managers

We have discussed a number of decentralized package managers. What is meant by decentralized is that most information about a package is contained within the package archive. There are package managers that take a different approach, a much more centralized one, where the information about packages is stored in a repository on the file system. A package manager like pacman needed to download a master package list to do its work properly. First of all, effort on the part of developers must be made to generate the master package list. Second, if users wish to find out about a package, they might need to download the package[60] and extract its contents. Apt uses a database of available packages, descriptions, and everything that is available if a package is extracted. All of the information is initially stored in a package. It can be argued that such a system is inefficient for both users and developers. First, copies of information are stored all over the place. Second, tools have to be developed and maintained that create copies of the data by automatically scanning available packages and package repositories.

Package managers exist, however, that eliminate the need for such duplication – which takes up space and cpu time – and that make development arguably more efficient and

[60] Or invoke a package manager to query, download, and extract appropriately.

organized. These package managers keep information about packages centralized. Note that this idea of centralized versus decentralized package managers is not a common idea, but a pattern identified by the author to categorize the vast number of package managers and package management systems out there.

The basic centralized package manager – from which the idea of centralized information gathering grew – is "ports". This is not an actual package manager, but rather a base for a package management system; we will think of it as a package manager because, package managers which utilize a ports system use the same basic concepts. Ports was initially implemented in a number of BSD-based systems and a number of new GNU/Linux distributions have adopted it as a base for their package managers.

The basic idea of ports is that it is a collection of files containing information about packages, including how they are built and patched, where they are downloaded from, descriptions, and dependencies. This collection of files is called a port. Ports vary in what they contain. Some may just have `make` files, patches, and separate description files. Others may have one centralized file containing all information. A package manager using ports usually works the same way as the package managers we have covered: it downloads and builds a program using the information provided by ports, installs it either to the system or first installs it to a fake root, optionally packages it, and moves the files to the system. A raw ports implementation is just about as tedious to use as Slackware's pkgtool. There is no dependency tracking or warnings about missing dependencies. Just about any ports implementation, however, has a way to quickly and easily update which packages are available. This is because the information is all centralized with a directory, which has subdirectories corresponding to ports. It is extremely easy for a developer to package the latest collection of ports and distribute it, perhaps as easy as writing a small command to be run periodically by a cron daemon. Furthermore, it is easier for developers to go to a central place to make any small changes, rather than having to repackage a program and wait for it to upload. With ports, developers can easily have a personal port collection, which they can use to test

how packages built with different options and add the build information – once they figure everything out – to the main ports repository. Everything in ports is designed to be built from source packages.

2.6.1 Portage

A bare ports implementation does not offer many advantages to the user. Ports based package managers have been, however, created that offer great advantages to the user. One such package manager is portage, which is used and maintained by the developers of the Gentoo distribution. Portage keeps a collection of ports on central servers to which users synchronize their machines whenever they wish to update their packages by using the `emerge --sync` command. The emerge application is part of portage and is a frontend for an underlying ebuild application. Ebuild and emerge work together to allow users to install, remove, update, and in two words, manage packages.

Portage's ports consist of a number of files. A `Changelog` file keeps information for developers regarding by whom, when, and why a port was changed. A `Manifest` file contains checksums for all of the files in the port with the exception of the `Manifest` itself. A `files` directory in a port includes a set of patches,[61] a digest file containing checksums for files which will be downloaded from the Internet upon package installation or update, files to be added to a package at some point, and a `metadata.xml` file which is used for special purposes to get information about ebuilds. Finally, the main file of a port is always the ebuild of the package.

Ebuilds contain all of the information about packages that portage needs to carry out its duties. Portage has become a very complex piece of software. The primary reason for this is that ebuilds are designed to be easy to write. If a package has a traditional `./configure`, `make`, `make install` installation procedure, the person writing the ebuild does not have to say in the ebuild that those three commands need to be run. Portage automatically figures

[61] This has recently changed; patches are now provided as separate archives.

out what to do in such a case. Portage also has something called an "eclass". An eclass allows a developer to specify what type of installation a package will have. Once portage sees an eclass declaration in the ebuild, it passes the package name to the eclass script or scripts which use that name to do what the package requires. This is especially useful if someone is dealing with a group of packages which differ in their installation only in what name is added to one place or another, such as a configure option. A basic ebuild starts out with an inherit declaration, which specifies which eclass to push the package through or which functions provided by portage to import so that the ebuild writer does not have to re-invent the wheel when attempting to, for example, compare program versions. Next come specifications for an assortment of variables, such as from where to download the package's sources, the description of the package, the license used by the package, the homepage of the package, which architectures the package can be built on,[62] build dependencies, runtime dependencies, post-merge dependencies, and any additional variables that the ebuild author wishes to use for his ebuild.

Before we describe ebuilds further, we must understand the categorization of dependencies that portage uses. Just about any other package manager that we have covered or shall cover does not categorize dependencies. In other words a dependency is just a dependency. Portage however utilizes categories. Build dependencies are dependencies required to build the package. Portage, due to the philosophies of the Gentoo distribution, tries its best to build and install all packages from source code on the user's machine. As a result, build dependencies are important for portage when it queues up packages to install. Build dependencies, specified by the DEPEND variable in an ebuild, will always be put at the front of the installation queue. Run time dependencies, specified by the RDEPEND variable, are dependencies the package will need in order to run. It does not matter where in the queue run time dependencies end up, so long as they end up in the queue. Post-merge

[62]Portage is built to be able to work on a fairly large number of computer architectures: http://packages.gentoo.org/.

dependencies[63] are those packages that are needed by the package, but need to be put in the queue after the package, because the package is needed to install the dependency. Post-merge dependencies do not commonly appear, and when they do occasionally appear it is because of USE flags.

Before we explain USE flags, we must finish describing what is contained within an ebuild. After variable declarations, a number of optional functions are added to the ebuild. If no functions are added, portage will handle the package as one that requires a number of traditional, non-specialized installation commands. Portage will guess and check through what it knows until it fails on everything it tries. There are a lot of available functions and all are required to be written in the order in which they are described.

A `pkg_nofetch()` subroutine can be written and used by an ebuild author to give instructions to users about what to do in order to download the source code if there is a package whose source code cannot be downloaded (usually for legal reasons). These instructions will be displayed to users when they attempt to install or update the package using emerge, and the source files are not downloaded. Next comes the `pkg_setup()` subroutine. This subroutine is used to make modifications to patches, make adjustments to variables depending on architecture, and do just about anything that might need to be done before the package is unpacked. Next comes the `src_unpack()` routine. In this subroutine the ebuild writer has control over what happens while and after the package is extracted by portage to its temporary build directory. It is useful, at this stage, to apply any non-standard patches that portage cannot handle. Some developers like to make sed substitutions[64] to adjust configure scripts, source code, etc. This subroutine also must be used if the package is extracted in a way that portage cannot recognize. The next subroutine is `src_compile()`. This gives the ebuild author the ability to control what options go into a package's configure script, in addition to specifying any commands that must be run before files are installed. The next subroutine is `src_test()`. Although we have not covered this

[63] PDEPEND variable.

[64] Sed is a program that uses regular expressions to search for and change text in files.

in our Linux From Scratch experience, packages occasionally come with scripts that test whether or not the package compiled correctly. This subroutine, seldom used, allows the ebuild author to specify whether a package should run any testing scripts.

The next function is `src_install()`. This function allows a developer to control what happens after a package is installed. A package installed by portage is never installed to the live system at first. It is installed to a fake root. Therefore, anything that the ebuild author does in this routine is done in the fake root. This allows portage not to have to make changes to its file tracking database before a package is ready to install. In other words, portage logs only the end result. In the `src_install()` routine developers make any needed or desired symbolic links, move documentation around, add things to configuration files belonging to the package, etc. Next comes the `pkg_preinst()` subroutine. At this point portage has built the entire package in a fake root, which can conceivably be packaged into a binary. The preinst routine allows the ebuild writer to complete any things that need to be done on the real system. These may include modifying configuration files of other packages, or more commonly, adding users and groups to accommodate a package. Next is the `pkg_postinst()` function. This routine is invoked after the package has been installed to the real system. It may include commands for linking needed things, but most commonly is used to display to users any post install instructions that the user must follow or information about how to use the package. The next function is `pkg_prerm()`. This function is used to do anything that might need to be done before portage begins removing a package. Likewise, the `pkg_postrm()` routine can be used to run anything after a package has been uninstalled. Finally, `pkg_config()` can be used to add things to configuration files after installation.

Within a lot of these functions, handling is done for things called USE flags. USE flags are options, named by ebuild writers, which allow a user to control what a package is built with support for and what it is not built with support for. If a user sets a USE flag, the ebuild will contain information regarding what to do with that use flag. For example, the flag

might force a certain DEPEND, RDEPEND, or PDEPEND to be added to the installation queue. Furthermore, the flag can set certain configure script options to be added in the `src_compile()` stage. Basically, ebuilds allow an easy way to organize a program to be customizable by a user without having to do research regarding dependencies and configure options.

Now that we have covered how portage's ebuilds are structured, we can cover how a user configures portage. In portage everything has a tendency to be centralized: from the ports collection underneath, to a configuration file with which users can customize portage to their preferences. Portage's main configuration file is `/etc/make.conf` where users specify the CFLAGS they wish every package to be built with. Because a number of users have abused their CFLAGS by adding too many of them, portage now filters CFLAGS from some packages. This is an option that is not possible with decentralized binaries. Next, a user can specify the CHOST, and CXXFLAGS, which by default are set to be the same as the CFLAGS. Before portage compiles a package, it sets all variables so that the packages' installations will use them. Next, users can specify a number of variables specific to portage. These variables include the directories portage uses, which may include a user's very own ports directory called portage overlay. The major variable that a user can set is the USE variable. Users set all USE flags which they wish packages be compiled with; i.e. if they set or unset a flag here, it will apply to every package that they install. For example, the nsplugin flag is used by packages to install integration with web browsers. If users wished that their web browsers had this integration, they would set this flag and not worry about any package that is updated or installed afterwards coming without integration with their web browsers. For example, the Real Media Player package can be made to work from firefox by setting the nsplugin flag and installing or re-installing both firefox and Real Media Player.

Portage also allows individual package flagging. Users can set packages to be masked,[65]

[65]This means that portage should not install these under any circumstances.

they can set USE flags individually, and do a number of other things that we will get to shortly. Before we finish describing portage, we must note that portage does all of the things that a basic package manager would do. It tracks files, it handles configuration files, and automates installing, updating, and removal of packages. These are all low level things and portage works on levels beyond them and levels beyond the basic ports implementation.

Portage by now might sound like a flawless package manager for both the user and developer. This is not the case. Portage may be extremely powerful, but is not perfect. Users still end up doing a number of things manually. For example, a user wishing to retain a specific, freshly deprecated version of a program will have to individually mask every single package which wants to bring in the non-deprecated version of the dependency. The question might be: why would a user wish to keep an old and rusty package? This might be due to Application Binary Interface changes that break source based systems. It might not be worth the risk – of rendering a system temporarily malfunctional – for a user to update a package that works correctly. Portage is also very big and bulky. Not only are there a great number of eclasses, which are difficult to document, but portage ends up doing a lot of inefficient things when going about its business due to its complexity. A number of Gentoo developers also complain about bugs in portage which have become features. These "features" were historically added to fix a bug, but now the "features" force ebuild writers to create workarounds because the "feature" fixed something, but broke something else. To remove such a feature is also not an option because it would bring back the bug that it fixed, or break existing workarounds. A number of Gentoo developers are developing a new package manager, called Paludis,[66] which hopes to work with existing ebuilds in the huge portage tree, do everything that portage does, but much more. Paludis is also developed to move away from some of portage's centralization and allow the execution of external scripts. Paludis also intends to create a feature where CFLAG optimizations can be added on a per package or per package category basis. Beyond this, Paludis wishes to

[66] http://paludis.pioto.org/.

extend portage's very limited ability to install into separate roots.[67]

2.7 Maintaining Package Managers

Now that we have covered all of the major package managers and package management techniques, we can proceed to cover levels even higher than the package managers. One of these higher levels is maintaining the package managers' packages or ports. Writing the code for the ports or manually building binary packages is a low level job. We have covered this. There are much higher level issues involved in managing the ports or binaries for a package manager.

2.7.1 Keeping Up To Date

The first of these issues is staying informed of when new versions of programs are released. Developers often need to keep up with what the latest versions of programs are so that they can provide users of the package manager the ability to utilize the latest software with the latest bug fixes, features, and improvements. There are more than ten thousand programs that are available to a GNU/Linux user. The most popular distributions have about that number of packages. Even with the hundreds of developers working to maintain packages for a package manager, they find it virtually impossible to keep track of all the packages. Many distributions assign individual developers to keep track of new versions and to package the new version into a binary or create a port for it. Even with this scheme there are problems because developers sometimes disappear, go on vacation, or do not have time.

[67] Recall swup's strong root specification feature.

2.7.2 ABI Changes

The next issue is related to ABI changes in programs. ABI is short for "Application Binary Interface". This interface is between the applications and libraries. In other words, if a library changes drastically, applications need to be adjusted in order to work. When a maintainer finds a new version of a program, he checks the program's changes log – if the program's developer had the courtesy to create one. If the program did not change much, then the maintainer copies most of the old port for creating the new one. If the package manager is not ports based, then the maintainer builds the new binary with the information contained in the source binaries of the older version. If, on the other hand, the program is a library and changed a lot, and the program provides functionality for another program, then the developer has a lot more work on his hands. ABI changes often require other packages to be rebuilt. Sometimes the changes result in new file names for libraries, in which case programs that depend on the old files' names need to be rebuilt so that they can link to the new library names. Making symbolic links is not practical because a large number of files might have changed their names and locations. On a ports based package manager, the maintainer submits the port for the new program and – if the port structure allows – passes a message to users telling them that they will need to rebuild programs x, y, and z, if they have them installed. With binary package managers, the maintainer rebuilds any packages that need to be rebuilt. This results in a problem, however, because the packages that are rebuilt cannot be used with the older version of the program with the ABI changes. Users will have problems if they decide to individually update one of the rebuilt packages, or more realistically, users install one of the rebuilt packages for the first time, without updating anything else.

2.7.3 Toolchain Weaknesses

The last big issue occurs when a maintainer stumbles upon a new package which does not compile on a specific toolchain for whatever reason. This can happen if the program

developer writes the program using a different toolchain. This issue does not affect users of package managers that use binary packages. In a ports implementation, however, the issue can reach the user. The toolchain that users have installed can vary on a ports system and their toolchains are required to build packages. Even if a maintainer checks a program on his toolchain, other users with different toolchains might have troubles, which present themselves in the form of compile failures. These failures can take a long time to fix depending on their nature. They often require patches to be created for the package, changing source code so that it can compile and work. Most of the time a maintainer is competent enough to create such a patch. However, if there are complexities, a bug is filed to the program's developer, and the maintainer waits for the developer to either fix the issue or report that it is some other issue. Sometimes the compile failures are fixed by patching a toolchain package. In this case, the maintainer and user will have to rebuild the toolchain package to get the new program to compile. The patch can cause regressions, however, so a developer might need to rebuild everything just to make sure that everything works. This is not practical and is done on a limited basis as a result.

2.7.4 Branches

As we have just seen, users have two options: stay informed and update their systems regularly or, don't install or update anything. Maintainers for package managers realize this and keep branches for the packages. The Debian distribution, for example, has three branches. An unstable branch, testing branch, and a stable branch. The unstable branch is where developers put the latest software with the latest rebuilds for ABI changes, the latest rebuilds with patched toolchain packages, etc. The testing branch is created from a snapshot of the unstable branch.[68] Testing is done from that moment to ensure that all packages work together correctly. Packages with ABI changes from that point are rarely

[68]This is done only when a testing branch becomes the stable branch and there is a need to start somewhere for a new testing branch.

added to the testing branch. The stable branch is made when a testing branch settles down. A testing branch settles down when maintainers are sure that all packages work together smoothly. Packages are updated in the stable branch, but only if they are known not to cause ABI mismatches.

Other distributions, notably Gentoo, have only two branches. A stable branch and a testing branch. If a package causes significant ABI mismatches, then the portage maintainers will mask the package until things get settled. Masking flags a package so that portage cannot add it to install or update queues. To use the package anyway, the user can manually unmask the package by editing appropriate configuration files. The testing branch is where maintainers usually commit ports as soon as they find out that a program is released. Portage, because it is a source based package manager, also has to watch for file collisions. In testing this is not watched, but for a package to reach the stable branch it needs to have no file collisions with other packages. Since Gentoo does not have a release cycle like Debian, the stable branch is not immune to ABI changes. The ABI changes, however, do not come to users as a surprise. When ABI changes happen in the stable branch, users are warned with a posting on the main Gentoo website and the popular Gentoo forums. When necessary, Gentoo provides documentation for how to properly update the system.

Here is a current example of a program with ABI changes and how distributions are dealing with it: `Xorg-7.1` is the set of packages which are responsible for rendering a graphical environment for users. It is one of the most popular sets of packages. This version came out with significant ABI changes. As a result of these changes all existing video drivers that provide three dimensional graphics will not work. The developers for the video drivers[69] are working on releasing ABI compatible video drivers. In the meantime maintainers have added all necessary ports and prepared the binaries. Gentoo put `Xorg-7.1` into the testing branch, and will hold off on adding it to the stable branch until new drivers are released and are tested. Debian put the latest Xorg package into the unstable branch,

[69]These drivers are closed source.

but will not touch the testing branch until the new drivers come out. Suppose however, that we have a user who does not care about three dimensional acceleration and the changes in Xorg will not affect him. If he is in a stable branch and using Gentoo, he will still end up waiting for the drivers to be made, unless he uses portage's "accept keywords" feature which allows the user to tell portage that a testing branch package can be used. A Debian user on stable or testing will have to wait.

There also exist branch-less distributions, which have only one, live branch. The biggest such distribution is Fedora Core. It stuffs the latest packages into the tree as quickly as maintainers can package them. Fedora relies on the fact that their users will keep as up to date as possible. In Fedora's case, a user would have to wait it out before updating the system again. Fedora, however, has a release cycle where they rebuild just about all of their packages from scratch and make a new release and eventually abandon the old one. Users usually cannot switch to the new release easily. We have to keep in mind, however, that Fedora is a distribution created as a playground for the more serious, non-free as in money, Red Hat Enterprise distribution.

2.7.5 Bug Reporting Systems

We have discussed probably half of the high level package manager maintenance process. The other half concerns the bug reporting systems. The open source community is how a distribution thrives. There are a number of for profit distributions, such as Mandriva and Suse, but they, like most other distributions, have a bug reporting system. Bug reporting systems are used for a large number of purposes. First, users eager for a package upgrade often report when a new version of a program comes out. More courteous users provide maintainers with information right in the bug report, such as the changes log. Even more courteous users provide working port files or package the source binary. Furthermore, users submit bugs relating to compile failures if they are using a source based distribution. In addition, sometimes maintainers do not notice an ABI problem until a user files a bug

report about something not working. Users inform maintainers of any typographical or documentation errors. Maintainers also file bug reports to keep track of what is happening. The bug reporting system allows maintainers to respond to important issues and make a distribution as pain free for users as possible. A maintainer does not even have to do tracking of packages coming out or ABI changes occurring. He can just sit back, relax, and do what the bug reports tell him to do. This is not a good way for a maintainer to think, but the idea is that he can focus on more vital things rather than have to worry about trivialities.

2.7.6 Optimization

Another high level issue floating around is the optimization problem. This is less of a problem for source based distributions,[70] because users can easily set their own optimizations. The only problem on source based distributions is that some users apply dozens of optimization flags and file bug reports when something breaks. This is more of an annoyance for maintainers, who turn down such users' reports telling them to tone their flags down to sanity. Binary based distributions, however, suffer from having to stick with one generic optimization level. This level is usually i386, for maximum compatibility for older machines. Once a binary based distribution chooses an optimization level, it cannot all of a sudden decide that it will release binaries optimized for another level. If it does this, a number of consistency issues with packages will arise, in addition to incompatibilities. For a distribution to go to another optimization level, its maintainers will have to repackage every package and release them all at once and abandon the older packages. This is no easy task and, as a result, distributions "stay put". Distributions like Arch Linux made themselves i686 optimized from their birth. However, the user base of these distributions has used i686 compatible machines from the beginning. A much older distribution such as

[70]Note that we use "distributions" rather than "package managers"; we are placing these together because once a distributions chooses a package manager, it sticks with it and is often known for it.

Slackware,[71] will irritate some of its user base, who use the distribution on old machines and would like to keep up to date, if it suddenly transforms its packages to i686. The next question might be, why can't a distribution release binaries for all levels of optimization? The answer is that it takes a lot of time to package a binary for more than one architecture. Another reason is that the distribution's server containing all of the packages will be filled with a ton of data, when instead it could be filled with one tenth that data if users could accept one generic optimization. Furthermore, benchmark results do not show much improvement with the higher optimization levels. See appendix A for a simple way of thinking about optimizations.

2.8 Frontends

We have introduced some frontends already, but we should deal with them in greater depth here. Frontends are a high level issue. A lot of distributions end up using the very packages that are maintained by either RPM[72] or dpkg.[73] This solves a lot of problems for distributions since they do not have have to worry about having to maintain a lot of packages. They do not have to recreate a collection of packages that took hundreds, if not thousands of people to bring together. The way these distributions set themselves apart is by creating a frontend for a package manager. A distribution like Debian creates two frontends for dpkg. One we have already discussed and the other is Aptitude. Aptitude, unlike apt, is specially geared for users just starting out with Debian. Aptitude does the sames basic things as apt, but has a more intuitive command line interface. Aptitude does not do all of the things that apt does. For example Aptitude does not allow users to search for individual files. This would conflict with Aptitude's search option, i.e. a second search option might confuse the user. Aptitude also limits the amount of control that users can

[71] It is the oldest distribution still active according to the Linux Weekly Newsletter's distribution list: http://lwn.net/Distributions/.

[72] Red Hat distribution.

[73] Debian.

have. It assumes that users wanting more control will use apt.

Red Hat[74] does not have frontends like Debian has. Instead, it uses a frontend developed by the Yellow Dog Linux distribution, called yum.[75] This frontend allows users to do the same things that apt allows and has Aptitude-like syntax, while allowing some of the control allowable under apt. As we have discussed, frontends make a user's life very easy due to the way dependencies and extensive updates of packages are handled.

Currently, distributions are setting themselves apart by creating GUI frontends for package managers. The distributions that are doing this are geared toward novice users.[76] There are two such distributions that have become popular. Ubuntu and Mandriva. Ubuntu is Debian based and comes with the standard apt and Aptitude. Their GUI is simple. It allows users to do a bulk update of all the packages. When a user turns the GUI on, the GUI automatically downloads package lists and everything that it needs. The user can also browse and install packages. The package manager interfaces with dpkg directly rather than relying on apt or Aptitude. Similarly, Fedora has GUI tools that enable users to install rpm packages that users might attempt to download via firefox; i.e. a user can open an rpm file with the GUI and install the package.

As we go up more levels, from low level package managers, to frontends, to GUIs, we appear to lose customizablilty. GUI creators often attempt to make things as simple as possible for the user. They assume that users do not know anything. As a result, they assume that users will not care about options that allow them to control things or they might get confused by all of the extra options and make an error. The GUIs currently in existence are therefore very limited in functionality, despite accomplishing very high level tasks.

We have now covered just about everything important about package managers, from the bottom up. We now proceed to implement a package manager of our own.

[74] Since Red Hat Enterprise Linux 5 and in Fedora.
[75] Yellowdog Updater Modified: http://linux.duke.edu/projects/yum/.
[76] By novice we mean users who do not really know much about their systems.

Chapter 3

The Birth of a Distribution: Developing the Package Manager

All of the package managers that we have covered in the previous chapter accomplish a wide variety of tasks and appeal to a wide variety of people. At present, it is difficult to come up with a package manger that offers much more than the existing ones. This difficultly aside, it is even more difficult to make a package manager that supports all of the software that is available. It took years and the work of hundreds of developers for distributions like Gentoo and Debian to contain over ten thousand packages in their respective package manager repositories. We, however, will attempt to overcome these difficulties and in the process see the type of thinking and work that is involved in creating a package manager.

The first stage of our endeavor has to start with planning which features we will focus on and how users will interact with the program. A user-first mentality has to be sustained throughout. We must also sustain an "advanced user" mentality by making it easy for developers and other advanced users to add packages to the package manager repository. The second stage involves writing the actual package manager and testing it along the way. The third stage involves extending the package manager and building a whole distribution

around it. There is no final stage because a package manager's repository must always be updated to accommodate the new developments in the large number of software projects.

3.1 Determining Features

Let us ask ourselves the following question: what can a user want that he cannot get elsewhere? Better yet, what does a user get elsewhere, which he can attain more easily? A possible answer can be: a package manager that supports both binary and source packages allowing the user to mix and match on demand or not to mix and match, but use the package manager the same way with both types of packages. To build source packages with rpm or dpkg one has to manually create build commands and then run the respective package managers with special CLI (command line interface) options to compile and install the source package. Even if the source package is acquired with build commands already, the user will still have to customize the build files for any special settings that might be required for compilation. Optimization flags, for example, are things that might need customization. In other words, such a package manager does not work well for source packages because of the extra knowledge the user must have and the extra steps the user must take. If a user wants a no hassle solution for building all packages from source, then he can go to something like portage. However, with portage the user will lose the ability to easily work with binary packages. With portage the user will have to first find a compatible binary package,[1] place the packages in a "magic" directory, and then run portage with appropriate CLI options.

To date, there are not many – if any – package managers that seamlessly work with binary and source packages. This is certainly a good feature that we might want to work on. Some packages take a long time to compile when building from source and users simply might not want to spend that time in hope of having a package optimized. They

[1] This is not an easy task considering that portage does not officially support a binary package repository and that not many people distribute binaries built by portage in the first place.

may only want their favorite media player optimized with special CFLAGS, while keeping everything else at the distribution's default optimizations. Furthermore, a user might want to get a specific system up and running quickly, and optimize it at a later time. Sometimes it might be faster for a distributor to create a port from which a package can be built from source, than for the distributor to create and distribute the binary package. In other words, a user will welcome the ability to choose between a system built from binary packages and from source.

Another feature users may want that they do not fully have at present, is a package manager that can seamlessly install multiple versions of the same package while being able to easily switch between all of the versions on demand. This is something useful for a user who wishes to test a package before removing the older, but functional version. Let us posit that we have users that use OpenOffice. Although the current version is 1.1.3, version 2.0.0 comes out with a number of new features that our users might find helpful. Our users are currently happy, but we know that they will not be happy if the new OpenOffice is broken when installed. The ideal feature of the package manager would be to detect the old installation and, upon installation of the new one, to backup all of the files that will be over-written before they are over-written. Then, if the users do not like the new OpenOffice the system administrators can, in one command, quickly backup – or remove – the files of the new version and resurrect the files of the older OpenOffice. Such a feature does not exist extensively in any package managers as far as the author knows. Some package managers, particularly portage, allow multiple installs for some packages. However, external scripts are required for users to be able to switch between versions because the packages are installed in separate locations and symbolic links need to be changed.[2]

The multi-versioning support can be expanded to feature something that we might want to call duplicate file handling, or "duplicate handling" for short. Recall the annoyance we described of packages that overwrite the files of other packages. Currently, package man-

[2]This is faster than the feature we are planning, but is much more involved for ebuild maintainers and less systematic in the sense that provisions must be made on a package by package, case by case basis.

agers either overwrite if the file exists or they tell the user that there is a file collision which must be cleared before continuing. Why not back up the files before they are overwritten for a particular package and resurrect the files if the package that "overwrote" the files is removed? This feature might not be noticed by the novice or uninterested users, but it would certainly solve a rather big headache for developers. This feature would be combined with an ability to use files belonging to a package on demand.

Another – and perhaps last – feature that we might want to implement is per-package and global optimization flags for packages. Currently, portage is king of package managers for users who wish to build a custom optimized distribution. This is partly due to the feature in portage which allows users to edit the main configuration file and set their custom CFLAGS, CXXFLAGS, LDFLAGS, and other optimization flags. However, in portage, users often run across the problem that some optimizations do not work well with specific packages. As we saw in the first chapter, optimization flags for package *x* can "de-optimize" package *y*. In fact, some optimization flags prevent a package from working altogether. Portage has a feature called optimization flag filtering. This feature allows a developer to designate harmful flags for individual packages. This is useful in the cases where a user sets global optimization flags that might break a package. The question becomes, why not expand this feature, and allow the user to set his own global optimization flags, but set his own package specific flags. Portage's current feature allows only filtering of flags for individual packages, but not for the setting of flags.

3.2 User Interaction

The first stage in developing our package manager is to think about how users will interact with the package manager. We must consider what CLI options users should have to work with and what they can change in the configuration file. We have to figure out what to name all of the commands and options. Our goal is to create a package manager that will

provide users with easy commands for doing common tasks. For example, if a user wants to install a package, he does not have to supply an `--install` flag as this is extra typing. Furthermore, a user should not have to supply an `--update` flag. The package manager should automatically determine if a package is being updated or not. A user should be able to check for and update all packages by simply supplying `everything` to the package manager. A user should rarely have to supply multiple CLI options since a single or no CLI options is more desirable. All of this helps the user by not forcing him to memorize things.

The first thing that we have to do is come up with a name for the package manager. It makes it difficult to refer to something as "package manager" all the time. Furthermore, it will help users if the package manager has a name. Let us call the package manager Vestigium, which is Latin for "trace". The basic concept of a package manager is to keep track of the files that are owned by packages. It makes sense to create the association between package managers and tracing. In production versions[3] of Vestigium it makes a lot of sense to allow the user to call both "vestigium" or "ves" at the CLI because it will save a user the trouble of typing more.

As we have already mentioned, users interact with the package manager in two ways: the CLI and the configuration file. The configuration file should serve the purpose of allowing the user to set settings so that the user does not have to use specific options on the CLI. For example, if the user always wants source packages, he can set it in the configuration file so that he does not have to supply a `--source`, CLI option. The configuration file should also allow the user to set the directories that Vestigium will use. For example, a user might have a partition designed specially for downloading source files. The user can make Vestigium download to that partition via the configuration file. The configuration file also serves as the place where global settings are set. For example, this is the place where users would set their optimization flags. Finally, the configuration file should be the place where users set their preferences of download mirrors and other things like this.

[3]Production versions of software are intended for use by the user. Production versions are those versions that are the result of extensive testing.

As mentioned briefly earlier, there should be no need for an install or update option. Vestigium should be able to automatically determine what to do depending on the information that it has. At this point we assume that Vestigium knows whether a program is installed and can correctly sort package version orders. Since Vestigium will not do any work if a package is already installed and up-to-date, a user needs to be able to force Vestigium to process the packages anyway. A user simply has to apply the `--rebuild` option to Vestigium's CLI. It is important to note that no provisions will be made for `--rebuild` in the configuration file because it would conflict with Vestigium's automated features.

When a user passes a package or list of packages, Vestigium will evaluate for the user whether or not any dependencies need to be brought in. If any packages have dependencies, it will add them to the front of the package installation queue. If a user desires not to have such functionality, he will be able to disable it by passing a `--nodep` option to Vestigium's command line or set the nodep flag to true in the configuration file. The nodep flag will not appear in the configuration file by default, because it is not a good idea for a user to set it to anything but false. We should, however, make sure that no user is deprived of the ability to do something he wants to do anyway.

Vestigium needs to keep its information about packages as up to date as possible. Therefore, it will automatically download all information that it needs. If the user desires to manually retrieve that latest information, he can set a `nosync`[4] option to true in the configuration file; furthermore, he can pass a `--sync` option to the command line. The `--sync` option will have two uses. If nosync is set to true in the configuration file, a user can retrieve the latest information by running Vestigium with the `--sync` option. If nosync is set to false, then `--sync` will not download the latest information. This may seem like unexpected behavior, but it will prevent the introduction of extra CLI option clutter and encourage users to make such changes in the configuration file. The reason for this is that a user must only prevent synchronization when he wishes to keep a static collection of ports

[4]Short for no synchronization.

or does not desire to wait for Vestigium to move on if it realizes that there is no network connection. Vestigium needs to be up to date so that maintainers can distribute any fixes to the users.

A CLI-only option for Vestigium will be `--download`, or `-d`. This option will enable the user to download everything that is needed without doing the actual installing or updating. This is handy for a user who might have a limited amount of time to download packages, and therefore desires to download everything that is needed for him to use when he does have access to the Internet.[5] The reason that this is a CLI-only option is that a user presumably will never make something like this a global option. This is because of the fact that the user will need to set the configuration file option back when he wants to update or install the downloaded packages' files.

Vestigium will have a `--debug` option, and a settable `debug` flag in the configuration file. This flag will be used to provide more verbose output to the user. It will show what subroutines Vestigium is entering and leaving. The debug `flag` is used heavily in development, and will probably be invoked by users submitting potential bugs with the package manager.

Another option that Vestigium will have is `--prompt`, or `-p`. This will print for the user a list of packages that will be installed, updated, rebuilt, or removed, and will ask the user for confirmation regarding whether or not to proceed. This option will be available in the configuration file and by default will be set to true. If the user uses `--prompt` with "prompt" set to true in the configuration file, he will effectively do nothing. The prompt option is useful because it allows a user to see all of the packages that will be worked on. The user can say "no" at the prompt and correct any mistakes.

Vestigium will have control options for removing packages. Vestigium will always correctly know whether a user is installing, updating or rebuilding a package. When Vestigium updates a new package it will, by default, remove the older package. The user can

[5] A user who for whatever reasons has access to an Internet connection for a short time will find this very useful.

change this behavior by either setting `--noremove` on the CLI or setting `noautoremove` to true in the configuration file. The `noremove` flag will not disable the ability for the user to manually remove packages or clean out dependencies. The user can remove a package by sending the `--remove`, or `-r`, option to the command line. Since removal is done on a package by package or list of packages basis, no such option will exist in the configuration file. We do not want to encourage users to remove packages whenever Vestigium is given a list of packages with no options.[6] The `--remove` option can be used by users with `--nodep` to prevent Vestigium from removing packages that were brought in as dependencies, but which will afterwards not be needed by any package.

Related to `--remove`, Vestigium will have a `--clean` or `-c` shorthand option, which will remove all packages of versions older than the given package. Recall that a major feature of the package manager is to allow multi-version installations. A user should have the ability to remove, with one command, all of the previous versions of a program.

Vestigium will feature a portage-like `--depclean` which will also have a configuration file flag `cleandeps` set to true by default. `cleandeps` will serve to search for dependencies that are not needed by any packages, and remove them. This is especially useful for users who run Vestigium with `noautoremove`, and want control over removing dependencies that are no longer depended upon by other packages.

As we mentioned earlier, one of Vestigium's prime features will be a special file collision handling system, called "duplicate handling". We make use of the term "duplicate" to help users understand that duplicates of files[7] are kept by Vestigium. By default, when a program is being installed and is attempting to write a file that already exists, Vestigium searches for which package the file on the system belongs to. Once it finds that package, it will back the file up under that file's backup directory.[8] The incoming file will get written once the existing file is backed up. The backed up file is called a duplicate, or "dupe" for

[6] This secures against the case where a user forgets about setting an option in the configuration file and ends up deleting important packages.

[7] To be more precise, the file names are the same. The contents of the files are probably different.

[8] By default /var/spool/vestigium/[package name].

short. Vestigium will have a number of options for handling duplicates. First, the user will be able to tell Vestigium not to create any duplicates, and just overwrite what is already there. This naturally can be done with a `--force` or `-f` shorthand option on the command line, and with an `alwaysforce` flag in the configuration file. The `--force` subroutine is intended to be understood by users as an option that overwrites everything in its path.

Second, the user should be able to prevent the incoming file from being written, and make it the file that Vestigium will back up as a duplicate. This is effectively flipping the files – or better yet, flipping the default behavior – and can be done with the `--flipdupes` option. The configuration file would have a `flipdupes` flag that accomplishes the same thing. If the flipdupes flag is set to true in the configuration file, then `--flipdupes` will naturally flip files back to their default locations; hence our justification for having "flip" in the name.

Third, a user might wish to remove duplicates from a particular package. He can accomplish this with the `--remdupes` CLI option. Because this is something that should be done on a package by package basis, no such corresponding flag should exist in the configuration file. We must keep consistent that if a user specifies a package without a version number, duplicates from all installed versions of the program will be deleted. Whereas if a package version is specified, Vestigium removes duplicates that are owned by that specific version.

Finally, a user might wish to apply a specific package's duplicates after installation. He can do this with the `--applydupes` or `-a` shorthand CLI option. This also is something that should not reside in the configuration file. The user will be required to supply a specific program with its version if the program has multiple versions installed. Otherwise, the program name will be sufficient. `--applydupes` makes sure to create duplicates for the program whose files are being replaced in favor of the other program. If used with `--force`, `--applydupes` does not create any duplicates. For example, package *x* owns file *a*. Package *y* starts installing and wishes to write its own file *a*. Vestigium copies file

a to package *x*'s duplicates directory, and then allows package *y* to write file *a* to the file system. The user does not like the file brought in by package *y*, so he uses `--applydupes` to bring in package *x*'s file *a*. Vestigium proceeds to save the current file *a* under package *y*'s duplicates directory and moves package *x*'s file *a* into the system. Had `--force` been utilized by the user, package *y*'s file *a* would not have been sent to package *y*'s duplicates directory.

Vestigium must provide some standard package manager viewing and searching features to users. A user might want to know what packages are available in the ports tree and what he has installed. He should be able to do this with a `--search` or `-s` shorthand option on the command line. By default the search will be shallow, in the sense that it will only search for package names. The user can pass a `--searchdesc` or `-S` shorthand option to the CLI, or modify the configuration file to have `alwayssearchdeep` set to true. This will search not just the file names, but the package descriptions. This is almost identical to the way portage does its searching.

Vestigium will allow users to search for files on the system to determine which package they belong to, if any. This will be done with the `--searchfile` or `-b` shorthand option. Since `-s` is already in use, `-b` is used instead, and can be associated with "belongs". Another thing that Vestigium will allow is the ability to list all files that belong to a package with a `--listfiles` or `-l` shorthand CLI option. Neither of these features needs to exist in the configuration file. Another searching option for the user will be `--verify`. This will check a package for whether the files in the logs exist on the root system. If there is a mismatch, Vestigium will tell the user which files are missing.

We have thus far only described how users utilize Vestigium's multi-versioning and duplicate handling features. It is now time to describe how users will take advantage of being able to choose between installing binary packages and installing source based packages. In Vestigium, this could not be any simpler. If it is set in the configuration file that the user wants binaries, then Vestigium will download and install only binary pack-

ages. If the user wants a package to be compiled from source, then he simply passes the `--buildsource` CLI option. The same principle applies if the user has source based packages set in the configuration file. The user, at any time, can install a binary package by passing the `--installbinary` CLI option. A user will not accomplish anything by running Vestigium with `--buildsource` or `--installbinary` while having source-based installation or binary package installation enabled respectively in the configuration file.

To extend the packaging capabilities, Vestigium will provide users with backup capabilities, whereby a user can pass `--backup` along with a package or list of packages to the CLI, thus creating an archive of files that are owned by a package. This backup will be stored in the user's backup directory which is chosen in the configuration file. At any time a user can recover the data saved in the archives by using the `--applybackup` option.

The package backup capabilities naturally allow for the creation of binary packages. Binary packages are, after all, archives of all the compiled files of a package. If a user desires to create binaries of packages as they are being installed, he can do so by running `--package` or setting `createbinaries` to true in the configuration file. If the user wishes to create a binary package without installing, he can use the `--binary` option or set `onlycreatebinaries` in the configuration file.

The last major feature of Vestigium is the setting of customization options. Global optimization options are set in the main configuration file. Users simply add lines like `cflags =`. For per package optimization flags, users add them to the package's INFO files in the package's port directory. Users who are not sure about their optimizations, or who want to receive optimized packages, can simply tell Vestigium in the configuration file – choosing from a list of known ones – what processor their system uses. There are two lines in the configuration file that users have to fill: the first is `arch =` and the second is `subarch =`. `arch` specifies the general architecture, that is x86, ppc, mips, etc. Subarch specifies the system in more detail and answers the question of whether it is an athlon64 32 bit system, a pentium4, or simply i386. Comments in the configuration file specify the

available archs and subarch options. Note that any optimization flags added by the user in the configuration file override the optimization flags brought in by the setting of arch and subarch. Furthermore, per-package optimization flags override all other optimization flags.

3.3 Implementation

Now that we have described how Vestigium will be designed to be used by end users, it is time to describe how package maintainers and other developers will use Vestigium. From the perspective of maintenance of Vestigium, the major consideration is to avoid a steep learning curve. This means that users should not have to learn a new language in order to create a build script. Build scripts should contain instructions for how the package is configured, how the package is built, where all of the files are installed, and what links are to be made.

The second consideration is expandability and organization. Vestigium will have a modified ports structure. A package's port, at least for now, will contain four files. The first file is the INFO file. INFO contains basic information such as a package's dependencies, the files that it needs to download for installing from source, a description of the program, the license used by the program, and a web link to the program's website. Next comes a BUILD file. This file contains the commands needed to configure and compile a program from source. After BUILD comes INSTALL. This file contains instructions for where files are moved in the system. Everything specified in INSTALL becomes logged by Vestigium; that is, all files that are installed get tracked by Vestigium. Finally, we have the POSTINST file where maintainers can add instructions to be run after a package has been installed, that is, any instructions that the maintainer does not want Vestigium to track. The reason for splitting files out is so that a novice maintainer can absorb the mentality of building the package first, copying files to the right locations, and then making adjustments afterwards. Portage's ebuilds force users to create subroutines for the three tasks.

The ports hierarchy will be designed such that all packages are in alphabetic folders. Inside each package's folder is a folder for each version of the package that is intended to be available. For example, the package screen-4.0.2 would be under the following path: `s/screen/screen-4.0.2`. The alphabetic portion of the hierarchy serves to accommodate a growing number of ports.[9] The package and package version portions allow for the creation of multiple versions of packages.

A utility, called Vesport,[10] will be available so that maintainers do not frequently have to recall syntax for the INFO file. Vesport is a very basic editor that asks a user to input data in response to questions. The questions revolve around creating a port for a given package. Once it collects all of the information from the user, Vesport creates the four files that comprise a package's port, and organizes everything with correct syntax.

The final consideration for developers and maintainers is that Vestigium and its accompanying tools are all written in Perl. Perl is a scripting language, which means that it is easier to learn and often easier to program in. Perl is considered by many to be a language that is difficult to read. This is certainly true if no effort is expended to make more readable programs. Vestigium is written to be easy enough to read by following consistent syntactic and semantic rules and having well named and placed subroutines, as well as comments. A developer or maintainer can very easily modify or add to Vestigium and its tools' functionality. A lot of package management deals with text processing.[11] Perl is considered to be one of the strongest languages for text processing, making it even easier to work with.

Now that we have explained how the package manager is intended to be used, we can proceed to describe the core implementation and development of it. We can look at what we described earlier as an outline for the implementation. The first thing we describe in the core implementation is the main code, that is, the code that can be seen as comprising the main method. This will help with the description of all of the subroutines that are used.

[9] It makes searching faster.

[10] Written by the author for use with Vestigium.

[11] File tracking can be argued to be text processing.

Next comes a description of the tracking methods, including what information is tracked and how. Strengths of the implementation are described after that. Finally, the weaknesses and lessons learned are discussed.

Vestigium starts out loading and setting variables from the configuration file. Vestigium is hard-coded to look into `/etc/vestigium.conf` for the configuration file. Most package managers also have hard-coded locations for configuration files, with CLI options that enable the user to provide the location of a configuration file. The main reason for hard-coding the configuration file location is that the configuration file contains information regarding the locations of everything else. Once the configuration files load, we do not have to hard-code any other locations. The other locations include the directory where Vestigium's libraries, ports, log files, backup directories, temporary directories, download directories, and wrappers are. In addition to locations, Vestigium extracts information such as optimization flags,[12] architecture and sub-architecture information, and the settings that we described earlier.

After loading information from the configuration file, Vestigium proceeds to make sure that the temporary directory is clean. As we shall see later, the temporary directory is used by Vestigium to build packages from source. When building packages from source with Vestigium, failures may occur in the middle, where Vestigium does not get a chance to clean up afterwards. Therefore, cleaning the temporary directory before anything else allows for two things. The first is that novice users can retry a failed package – or continue to use Vestigium after a failure – without having to know about the potential need to delete information from the temporary directory. The second is that advanced users, who are attempting to create a port for a package, do not have to waste time remembering where the temporary directory is and deleting its contents.

The next thing that Vestigium does is load sub-architecture default optimization flags. Vestigium is designed so that users who want an optimized Pentium IV system can get

[12] As of this writing Vestigium is coded to support LDFLAGS, CFLAGS, CXXFLAGS, and CHOST.

one without having to read the gcc manual themselves. That is, they can trust Vestigium's maintainers with that job. Vestigium's maintainers are supposed to set optimization flags in a special file located in Vestigium's library directory. This file contains a list of known subarchitectures[13] with optimization flags corresponding to them. Vestigium makes use of Perl's regular expression capabilities to efficiently extract all of the optimizations. The way information is stored in the subarchs file makes it easy to understand for those intending to modify it and faster for Perl's regular expression engine to read.[14]

We have discussed before how optimization flags added to Vestigium's configuration file override those set by the choice of the subarch. In order to correct this, Vestigium takes those extracted optimization flags and replaces those set right before.[15] Note that optimization flags are placed into environmental variables. Perl provides easy access to the environment variables by allowing programmers to edit directly the ENV hash. This means that we can easily override the optimization flags that we set earlier without needing to keep track of any extra variables. Furthermore, almost all programs that we will compile with Vestigium only need to have the environment variables set. If the user did not specify any custom optimizations, the optimization variables will be empty because the flags extracted from the configuration file went straight into the environment hash. Therefore, Vestigium makes sure to set the optimization variables to what is in the environment hash.

3.3.1 Libraries

The next thing that Vestigium does is load the libraries. At the time of this writing, there are three libraries that are imported. The first is `search.lib`. This library contains subroutines needed to search for files, owners, and programs. The subroutines include `searchBackupsForFile`, which searches for a file in the duplicates directory of all packages. When found, the location of the file is returned. From this location, the program

[13] Such as Pentium IV, Pentium III, Athlon64.
[14] See the subarch file in the appendix and take note of the indentation
[15] If the user did not set flags in the configurations, this will not occur.

that owns the file can be extracted. This is due to the directory hierarchy that we discussed earlier. Another subroutine is `findOwner`, which finds the program that owns a given file. Another subroutine, `findDupeOwner`, finds all of the owners of a file in the case where there were file collisions and files had to be moved out to duplicate directories. Another routine is `findProg`, which, given a program name, returns all of the versions available for installation. Next is `findInstalledProgVer`, which, given a program name and version, returns whether it is installed or not. Similar to it is `findInstalledProg`, which returns a list of all installed versions for a given program. Next is `listFiles`, which checks if a program is installed, and if it is, returns all of the files that the program owns.[16] A small subroutine exists to output things that are not found. Called `notFound`, this subroutine is used mostly by the other subroutines. It accepts what was not found as an argument, prints out a warning about it, and goes on to call the `continuePrompt` subroutine. Last, but not least, a simple function, `checkIfDuplicate`, is used to check a given list in Vestigium for an existing entry. This is used later in calculating dependencies.

The next library is `cont.lib`, containing three subroutines. The first is the `printHelp` subroutine. It is called every time a user supplies syntactically incorrect CLI parameters to Vestigium. Vestigium's CLI syntax is simple. A user is expected to provide, at the bare minimum, a package or list of packages to process or `everything`, which for Vestigium means all packages. A user is also expected to provide correct spelling for any CLI options. Help is displayed if information is missing or deficient in either case. The next subroutine is `error`. It takes as a parameter an error message and returns the error message while stopping Vestigium in its tracks. The `error` subroutine can be replaced by the `die` function in Perl.[17] However, it is a convenience for those reading Vestigium's source code to see a subroutine named `error`. The `error` subroutine is used in Vestigium predominantly for sorting out contradictory CLI options. For example, if a user uses the CLI options `--search` and `--remove` at the same time, we need to stop Vestigium and tell the user

[16] File ownership is defined as the files that a program has installed.

[17] error uses die.

why.

The last library is `names.lib`, which provides subroutines that improve Vestigium's readability. The first subroutine is `getProgName`, which, given a program name with version, extracts the program name. The second is `getFirstLet` which returns the first letter within any text. And finally, the last is `getBasename`, which returns the last directory in a path. All of these subroutines consist of simple regular expressions.

After loading libraries, Vestigium continues by checking if the current user is the root user or not. This is done because installation is made to directories that only root can write to. However, since it is not necessary to be root in order to search for file owners and similar things, Vestigium prompts the user to continue or exit. The continue prompt simply asks the user to continue, and waits for their input.

3.3.2 User Requests

Vestigium proceeds to load input from the CLI. It makes sure not to accept duplicate entries of options or of packages. If the user inputs: `vestigium glibc glibc`, Vestigium will not work on glibc twice. If the user supplies `everything`, Vestigium will work on all *installed* packages. Vestigium knows which packages are installed by looking into the log directories. The rest of the loading input from the CLI consists of setting all of the variables necessary for Vestigium to work. After setting all of the variables, Vestigium does a comprehensive sanity check to make sure that users are not abusing the CLI with contradictory options.

If Vestigium passes the sanity check, it proceeds to synchronize the ports, unless the user specified otherwise through the CLI or configuration file. The synchronization of ports is called with the sync subroutine. It looks for download mirrors specified in the configuration file, and then downloads an archive of ports. The download mirror is the general location of where to download files from. The mirror must contain a special hierarchy. There has to be a directory named ports, within which a `ports.tar.gz` file is located. Within this

directory is the hierarchy of packages that we described earlier and `ports.tar.gz` is in this directory, because it is easiest for the maintainer of the mirror to create archives there. It is faster to create and faster to extract `ports.tar.gz` because it is a `gunzip` archive.[18] The idea is that every time there is a change to a port, a new `ports.tar.gz` archive is created and made ready for Vestigium to download. Once Vestigium downloads it to the download directory, it extracts all of the files and removes `ports.tar.gz`.

If the user wanted to use `--search` or `--searchdesc`, Vestigium will do that by reading all of the packages provided by the user on the CLI. For every package, Vestigium will call the `findProg` subroutine, which will or will not output – depending on whether the program exists or not – a list of all available versions of the program. The subroutine will take care of notifying the user if a package was not found. For every available program version, Vestigium will check whether that version is installed by calling `findInstalledProgVer` and will notify the user of what it found. At the time of this writing the `--searchdesc` method is not yet implemented. In order to implement it, one would call a new subroutine, but only after attempting to find programs of a certain name. The new subroutine would scan all of the descriptions in all of the ports for a specific keyword. Of course, this would be called by specifying it on the CLI or in the configuration file.

If the user told Vestigium to remove duplicates for a given package or packages, Vestigium would do this now by figuring out if the packages supplied are installed. It makes no sense to go ahead and waste processor cycles and hard-disk access times to do something that is already done. If a user wishes Vestigium to proceed with deleting a package's duplicates, he can say yes at the continue prompt. Vestigium will always remove duplicates of the latest installed version, unless the user provides an explicit version. The `findInstalledProg` subroutine takes care of versioning. The removal of duplicates is accomplished by the `removeDupes` subroutine. The `removeDupes` subroutine simply extracts the information necessary – from the program and its version – to find the path of the dupli-

[18] A gunzip archive is larger than a bz2 or bunzip archive, but speed is more important here.

cate files, and proceeds to delete everything in that path. For example, glib-2.4's duplicates would be located in `$backupDir/g/glib/glib-2.4/dupes/`, where `$backupDir` is the location of the backup directories which are set in the configuration file.

The next option that Vestigium might work on is outputting a list of all files owned by a package for the user. This is done by calling the `listFiles` subroutine for all packages given by the user. The subroutine handles all of the issues relating to informing the user whether packages are installed, do not exist, etc. Similar to the list files option is `verify`. If the user invokes `verify` the difference is that it does not output the list of files to the user, but takes that list and scans the disk for the existence of every single file. The `listFiles` subroutine collects its data from log files left by Vestigium when installing the packages. This means that all of the files in the logs should, under normal circumstances, be in their appropriate locations. If this is not the case, the `verify` option outputs which files are missing.

3.3.3 Queues

If Vestigium is still running it is because its attention was not grabbed by the users' requests to do specific actions. Vestigium now needs to build what we will call package queues. Depending on what the user wants, queues need to be constructed so that Vestigium can apply the correct subroutine to the correct packages and in the correct order. First, we need to describe all of the queues that Vestigium works with. First is the uninstall queue. Packages are added to this queue – at the time of this writing – if a user passes the `--remove` flag to Vestigium on the CLI. Other cases where packages might be added to this queue are more complicated, and are not yet implemented in Vestigium. These cases include the updating of packages, if the user supplies the `--clean` or `--depclean` option. Both are difficult to implement, because they have to scan all installed packages for whether a particular package is depended upon or not. This is especially difficult because in its current state, Vestigium cannot distinguish between packages that were compiled as dependencies

and those that were not. A simple distinction is made by the Portage package manager, but even then, `--depclean` is still not efficient. Portage maintainers do not trust their scheme to be used during an update as a way to remove obsolete packages. The simple distinction can be made between those packages that historically had to be added as dependencies and those that the user explicitly requested. This is the distinction Vestigium would probably have to make if someone were to implement the `--depclean` feature along with the ability to track down and remove obsolete packages, or even switch dependencies.[19] The `--clean` feature is also not implemented as of this writing. This is due to the fact that it is not a major feature and can be done without for some time.

The next queue is the backup queue. Just like uninstall, it responds to the `--backup` option supplied by the user on the CLI. Unlike uninstall, however, it is not intended to be used for anything else. We nevertheless include it as a queue instead of just proceeding directly through all of the given programs. This is because someone might potentially find a use for it and end up having to do less modification to implement a feature.

Another similar queue to backup is the rebuild queue, which holds packages when a user supplies `--rebuild`. Packages will also be added to the rebuild queue if a user's configuration file says that the user wants to use only backed up packages. In other words, backed up packages will be installed regardless of whether they are already installed or not and if the user wants to work with backups only. We will see later that the rebuild queue and install queue use the same functions for dealing with backups.

The next queue is the install queue. Install gets populated with packages that are not yet installed. The latest version of available versions is added to the install queue unless the user specifies a specific version to install. The same goes for the update queue, except that packages are added to the update queue only if an older version is already installed. What occurs is that packages which are given by the user are checked with the `findProg` sub-

[19]For example, xpdf was a dependency for package *y*, but along come poppler with better support for package *y*. An implementation can be made where the dependencies are switched in the sense that xpdf is removed and poppler installed.

routine. This subroutine gives the list of all available packages. We take the latest version from this list and compare it to the output of `findInstalledProg`. If there are no installed versions of the program, we place the program into the install queue. If there are installed versions and the latest installed version is not the same as the latest available version, we add the package to the update queue. Otherwise, a package would be a candidate for the rebuild queue, but only if the `--rebuild` option was passed. Note that, had the user passed a specific version of a program, `findProg` and `findInstalledProg` would have returned entries stating that an available version was found,[20] but an installed version was not.[21] This would place the specific version into the install queue, as we would want, preserving the multiple installation feature that we outlined earlier.

The last queue is a temporary one because it gets scoped in and out in Vestigium.[22] The queue is called `depQueue`, and serves as a queue that collects dependencies that get added to the update and install queues in the right order. The mechanism for getting this to work is fairly tricky. First, Vestigium cycles through what is in the install queue. If the user did not set `--nodep`, it continues with the process. Vestigium extracts dependency information contained within a specific package's INFO file. It creates a list of dependencies whose existence must be checked. If the latest version of a dependency is already installed, then nothing needs to be added to any queues. If a dependency is not installed or is not the latest version, Vestigium adds the dependency, along with its latest available version, to `depQueue`. Since different packages might have the same dependencies, Vestigium does not add duplicate entries of dependencies into the dependency queue. The question might be at this point: why do we not add the dependencies directly to the install, rebuild, or update queues? The answer is that dependencies might have dependencies themselves. If we add directly to the queue without using `depQueue`, we will have to recheck all of the queues for dependencies. Even worse, we will have to do another iteration, followed by another, and

[20] This is assuming that the user supplied an existing program with version.
[21] This is assuming that the program version was not installed.
[22] The queue is alive in only a small section of the code.

another. Instead of that trouble, a recursive algorithm is utilized for finding the full tree of dependencies in `depQueue`. For every entry in `depQueue`, the entry's INFO file is parsed for dependencies. Vestigium checks if all of those dependencies are installed or not and if they are the latest versions. If not, the dependencies are added to the back of `depQueue` – this is crucial – if a dependency is not already in the queue. Vestigium then does the same thing to the next item in `depQueue`. Since all additionally added dependencies were added to the back of the queue, Vestigium eventually reaches them and finds their dependencies. What we have at the end is an ordered list of dependencies such that the core dependencies – the ones that others depend on – are at the back, while the ones that need dependencies are at the front. This allows Vestigium to easily add everything in `depQueue` to the install queue and update queue. Vestigium does this by checking if the dependency has an older version already installed or not. In the former case, the dependency is added to the front of the update queue. In the latter, the dependency is shifted to the front of the installed queue. Note that Vestigium takes dependencies from the front of the `depQueue` and moves them to the front of the other queues. This is important, because at the end the packages that do not need special dependencies are at the front of the queues, while packages that do require dependencies are at the end of the queues. Without this order there would be trouble, especially if the user is building packages from source, since packages will not pass their configure tests and will not allow Vestigium to continue. Another thing that we have to note is that circular dependencies are possible with the algorithm in its current state – as of this writing. However, circular dependencies might be abolished from Vestigium by comparing duplicate entries from `depQueue` before they are added to the other queues.[23]

Vestigium now deletes `depQueue` and proceeds to inform the user about what it is going to do, unless of course the user requested in the configuration file not to be prompted. If the user tells Vestigium to continue, it will proceed to process every package in every queue. First is the install queue. If the user specified in the configuration file that he wanted source

[23] See Appendix B for more about circular dependencies and methods of combating them.

packages, Vestigium runs the `installpkg` subroutine. If the user specified that he wanted binary packages, then Vestigium will run the `installbin` subroutine. And finally, if the user specified that he wanted backups to be extracted, then `installbak` will be run. All of these subroutines are covered later. The same applies with the rebuild queue. The only difference is that the `rebuildpkg` and `rebuildbin` subroutines are run respectively, and the subroutine for backups is the same as for the install queue: `installbak`. The update queue is handled in the same way as the rebuild queue was, with the exception that the first two subroutines are once again renamed. The last two queues, uninstall and backup, have the `removepkg` and `backuppkg` subroutines run respectively. Once all of the contents of these queues are processed, Vestigium stops.

3.3.4 Subroutines

Up to this point Vestigium has not done very much. Most of the work occurs in the subroutines. We will not describe all of the subroutines in detail.[24] The subroutines developed and well tested at the point of this writing deal with many of the same concepts and algorithms as other subroutines. To spare being repetitive we focus on the major ones: `installpkg`, `backuppkg`, `installbak`, `removepkg`, and `rebuildpkg`. These subroutines utilize other subroutines, which we will cover as we proceed.

The `installpkg` subroutine is given a package with its version that is not already installed. The first thing that `installpkg` does is open the INFO file for the package and checks if any optimizations flags were supplied. If optimization flags were supplied, Vestigium overrides the existing ones in the environment hash. The next thing that is done in `installpkg` is the calling of the `extract` subroutine. The program to which work is being done is passed to `extract`. The program's INFO file is first loaded by `extract` and used to determine the locations where the files containing the given package's source code can be downloaded. It takes the list of locations and calls the `download` subroutine. The `download`

[24] Not all of them have as of this writing been well tested, and some of them have not been implemented.

subroutine takes a URL as a parameter and downloads what is in the URL to the download directory. The `download` subroutine stops if the user supplied `--download` to the CLI. Vestigium makes sure not to download a file that is already in the download path by checking if the file to be downloaded is already located at the appropriate location. Once a file is downloaded or determined to be in the correct location, `extract` attempts to determine what type of file it is. Most of the files that circulate in the open source world are `tar` archives compressed with `bunzip` or `gunzip`. The `tar` command handles both of these cases by extracting the archives to the temporary directory. Another type of file that circulates is a patch file. Such a file usually contains information regarding what line to modify in the extracted source code. This file, with its location, is added by `extract` to a list of patch locations. The `extract` subroutine is designed to be extensible to include handling for any file format in existence provided that Vestigium's maintainer adds a line in the code for every format. After extracting all of the files, `extract` changes the current directory to the first extracted folder. This is called the working directory and theoretically provides the commands in Vestigium access to everything that was extracted in other directories. This is very convenient because users can write BUILD scripts where files are copied from directories adjacent to the working directory using something like `cp ../otherFolder/file file`. The last thing that `extract` does is execute `applyPatch` on all of the patches that were downloaded. The `applyPatch` subroutine attempts to patch the source code by passing possible options to `patch`. The `patch` command includes a feature that removes the parent paths from patches, allowing `patch` to patch the correct files in the source code.[25] Depending on who created the patch, the number of parent paths that need to be removed might differ. Vestigium figures this out for the user. If patching fails, Vestigium stops. Once patching is stopped, we are back in `installpkg`. The first stage of installation has finished and the source code is ready to be configured and compiled. At this point Vestigium generates a file with information about the package that is currently being worked on.[26] Vestigium loads

[25] See the patch program's man page.

[26] We describe this in detail when we get to talking about the file tracking method.

the package's BUILD file and begins running the commands that are contained within it. If the command is a cd command,[27] Vestigium updates the working directory to it. This is to ensure that when Vestigium goes on to the other scripts, it will not inadvertently lose the directory in which all of the action is supposed to take place. Once all commands in the BUILD are executed, Vestigium prepares to load the INSTALL file by creating the directories needed for the package's log files. Next, `installpkg` modifies the environment PATH variable to have Vestigium's wrapper path take precedence over other executable paths. We will discuss what this is later. All of this is necessary for correct tracking of the files that package installs. Without going into too much detail at this point, `installpkg` runs the commands in INSTALL so that all files that the package installs to the system are logged to files in a package's log directory. Once all commands in INSTALL are run, Vestigium resets the environment's PATH variable to what it was before and proceeds to run the command in the POSTINST file. The last things that are done in `installpkg` are the resetting of optimization flags to what they were before the package's optimization flags over-rode them and the removal of the contents of the temporary directory as well as the information file generated earlier. If the user had used the `--package` CLI feature or set its equivalent in the configuration file, Vestigium would call `backuppkg`.

The `backuppkg` subroutine is Vestigium's subroutine for creating binary archives of packages. The `backuppkg` subroutine, unlike `installpkg`, takes as input a program name with its version or a program name without a version. Assuming that the program is installed on the system and has correct information in its logs, `backuppkg` calls `listFiles` to get a list of files that are owned by the package on the system. This list of files is fed to the `tar` command so that all of the file's attributes are preserved – including symbolic links. The resulting archive is a `gunzip` containing all of the files of the package. It is important to note that the resulting file name will be the program name, followed by version, then architecture, and finally subarchitechtures. This allows us to create archives that can easily

[27] Change of directory.

be used to create a mirror with binary packages for users to download.

The `installbak` subroutine extracts existing backups of a package. When extracting the backup, log files are recreated. Vestigium captures the verbose output of the `tar` command to quickly create logs. The `installbin` subroutine completes almost the same actions that `installbak` does except that it downloads binary packages from a mirror on the Internet. Likewise, the `rebuildbin` subroutine completes the same actions, but makes sure to remove a package's files from the system first. The `updatebin` subroutine simply calls `installbin` on the new version and `removepkg` on the older version of the package.

Another subroutine is `rebuildpkg`. It first moves log files out of the way, making it seem as if the package is not installed. Next, it sets the force variable to true and calls `installpkg`. This makes `installpkg` over-write all files that will still exist in the system since the package is already installed. Once `installpkg` finishes, `rebuildpkg` sets the force variable back to false. Next, `rebuildpkg` compares the log files that were moved out to the new log files created after re-installation. If it notices any files in the old logs that are not in the new logs, `rebuildpkg` will remove extra files found in the old logs. Although it has not been implemented at the time of this writing, `updatepkg` will be almost identical to `rebuildpkg` with the exception that it will not need to copy log files to other locations, but compare the new and old versions and delete the files only associated with the older version.

The next subroutine is `removepkg`. This is one of the trickiest parts of Vestigium. Like `installpkg`, a lot of this subroutine depends on the file tracking implementation that we have yet to describe. The `removepkg` subroutine provides half of the implementation behind the duplicate file handling and multiple versioning features. `removepkg` first checks if a program is installed. If not, `removepkg` warns the user and prompts him to continue or not. Note that `removepkg` finds out if a program is installed or not by calling `findInstalledProg`. Had the user told Vestigium a specific program version, `removepkg` would process that specific version only. If a program is supplied without a version at-

tached, `removepkg` will attempt to remove all installed versions of a package. The first step to removing a package is scanning the two types of log files that were created during installation. The first type of log file is `inst.log` and the second is `dupe.log`. The `inst.log` file contains all of the files installed by a package. The `dupe.log` file contains all of the files that initiated duplicates for other packages.[28] Under ideal circumstances there will be nothing in `dupe.log`. In this case, `removepkg` can remove every file contained within `inst.log`. However, under circumstances where files were moved to a package's duplicates directory, `removepkg` needs to check if the files it is about to remove are in the program's duplicates directory. If they are, `removepkg` cannot remove those files because they are owned by another package. Furthermore, some entries in `inst.log` might be directories. Directories are removed by `removepkg` if and only if they are empty. It is important to note the Vestigium goes through the logs backward, starting with the last entry. It is designed to do this because during installation directories are created first, and then populated with files. When removing a package files have to be removed first, then the directories.

The `removepkg` subroutine is still not finished. It needs to take care of the case where it might delete a file that the package itself forced out. In other words, during installation a package might have forced some files to go into other packages' duplicate directories. For example, package *x* wrote file *a*, and then package *y* wrote file *a*, forcing *a* to go to package *x*'s duplicates. Next, package *z* also comes in to write file *a*, thus forcing file *a* also to go under package *y*'s duplicates. If the user decides to remove package *z*, removal of the file proceeds because package *z* does not have file *a* in its duplicates. The problem now however, is that package *y* and package *x* expect that file *a* is on the system. What `removepkg` has to do is find anyone with file *a* in its duplicates directory, and move that file from duplicates to the system directories. The `removepkg` subroutine does this by checking every file that has been recorded in `dupe.log` and attempting to find the other owners of

[28] Other packages' files had to be moved out before installing the new package's files.

the file. The first owner of the file that is found, and who has the file in their duplicates directory, is the one whose duplicate file is moved back to the system. If no owners are found, a warning is sent to the user. It is possible that all packages who had duplicates for a given file might have already been uninstalled, however if this is not the case then the user should file a bug with Vestigium's maintainers. The last thing that `removepkg` does is remove a package's log and backup directories.

3.3.5 File Tracking

Up to this point we have have had a few glimpses at Vestigium's file tracking implementation. It is time for a comprehensive description of the implementation. Vestigium's file tracking implementation is similar to the LD_PRELOAD method described in the second chapter. The main difference is that Vestigium's implementation is far more portable. In general, Vestigium utilizes the output from commands that are responsible for moving, copying, and linking files. That output is processed so that any file that is created can be logged in a log file or files. There are five main commands that are used in a package's installation to `make` files appear in locations on a file system: `mv`, `cp`, `mkdir`, `ln`, `install`. The `mv` command moves a file from one location to another. The `cp` command does the same thing execpt it does not delete the old file. The `ln` command creates links from a specific location to another file. The `mkdir` command creates directories. And finally, `install` is a versatile combination of the `cp` and `mkdir` command that can set permissions, strip symbol tables, and do other tasks. The author tried his best to find packages that used any commands other than these five during installation, but was not able to find any.

All `cp`, `mv`, `ln`, `mkdir`, and `install` commands that are run in INSTALL need to be run through Vestigium's modified scripts that log what those commands do. Commands in INSTALL, however, can be scripts such as `make install` or `sh program-install-script`. This means that the commands in those scripts must run through Vestigium's modified

scripts. The way Vestigium takes care of this is by having scripts called wrappers whose names are `cp`, `mv`, `ln`, `mkdir`, and `install`. These wrappers are located in a directory, which the `installpkg` subroutine shifts onto the PATH environmental variable so that the wrappers have precedence over the normal commands. With this precedence, whenever an installation script uses one of the five commands, it ends up running the wrappers. The wrappers execute the normal `cp`, `mv`, `ln`, `mkdir` and `install` commands. Initially, the wrappers would evaluate the input of the commands, and piece together the source files, destination files, and CLI parameters, and feed them into the normal commands. This turned out to be a daunting task, so instead, the author utilized the commands' verbose outputs. Here for example is the verbose output of the `cp` command:

```
# cp kde-andrey/* /tmp/ -Rv \par
‘kde-andrey/konqueror.log’ -> ‘/tmp/konqueror.log’
‘kde-andrey/ksycoca’ -> ‘/tmp/ksycoca’
```

The first item after `cp` is the source files and the second item is the destination location for those files. The `-R` option tells us that we want to copy directories and `-v` tells us that we want verbose output. The two lines after the command tell us what the `cp` command accomplished. It moved the file konqueror.log to the `/tmp/` directory, and did the same thing for ksycoca. Similar output exists for the `mv`, `install`, `ln`, and `mkdir` commands.

The wrappers do much more than just read the verbose output of the five commands. They log all of the information to the log files of the program being installed, and they move out any existing files before allowing any of the five commands to overwrite them.[29] Let us first describe the `cp` wrapper. We will then describe a small portion of the `ln` wrapper. The `ln` wrapper is almost identical to `cp`, with the exception of one interesting case. Next, we describe the `mkdir` wrapper. The `mv` and `install` wrappers are not discussed because they are nearly identical to `cp`. By going through descriptions of the wrappers we will see the full implementation of Vestigium's file tracking system.

[29]This is part of duplicate file handling feature.

One of the first things that the `installpkg` subroutine does is create a special file containing information about the package that is being worked on. This file, called the profile, is the first file that our wrappers read. The contents of this file are: the name of the program with the version, the directory where Vestigium's libraries are located, the location of the log directories, the location of the backup directories, a statement if debugging is enabled, a statement if the `--force` feature is enabled, and finally, the location of the work directory. The `cp` command – as well as `mv`, `ln`, and `install` – has a neat backup feature whereby the user can tell `cp` to copy an existing file out of the way before copying a new file over. There are two environment variables that can be utilized to customize `cp`'s backup feature. The first is VERSION_CONTROL, which tells `cp` the type of backup naming scheme to use. The choice is between simple and numbered. The numbered scheme does not overwrite files that have already been moved out of the way. The simple scheme overwrites any existing files that were copied out of the way. Vestigium is designed to use the simple scheme because, as we shall see, the numbered scheme is of no use. The second environment variable is SIMPLE_BACKUP_SUFFIX. It controls the suffix that is appended when moving files out of the way. To make everything consistent and more understandable in Vestigium we set the suffix to be `.dupe`, meaning that the file became a duplicate. After setting the two environment variables, `cp` proceeds to read the arguments that were supplied to it. That is, these arguments are the same arguments that a normal `cp` command would take. The `cp` wrapper takes off certain CLI options if supplied. The first option that is taken off is `-b` or `--backup` as it is intended for nothing other than the `cp` wrapper to control this option. The second option that is chopped off is `-S` or `--suffix`. This option controls file backup options, which might interfere with correct functions of the wrapper. Next is `-v` or `--version`, which, once again, only the wrapper should have control over. Finally, the `-V` or `--version-control` options that are used by `cp` commands on some systems, are filtered out to assure correct functioning of the wrapper. These options control how file backups are made. Once these options are filtered out, the wrapper changes the

current directory to the work directory. This is a measure taken to prevent the occurrence of a `cp` command assuming that it is being run from the work directory, but in fact is run in the directory from which the user is running Vestigium. After this step, the wrapper runs one of two variations of `cp`. In the first variation `--force` is used. In this variation `cp` overwrites any files that might be in the way. In the second variation, `cp` is run with the `-b` CLI option. The `-b` option is short for `--backup`. It tells `cp` to move the file that would normally be over-written to the same directory and append the suffix supplied by SIMPLE_BACKUP_SUFFIX. Both variations of `cp` are run with the `-v` or `--verbose` CLI option. For both variations, the wrapper swallows the output of the commands, including error output. The next action taken by the wrapper is the evaluation of the output that it swallowed. If the output shows that a backup operation has occurred, the locations of the backed up files are recorded in a duplicates list. In all cases, the file that was successfully copied to a destination is added to the installed list. In the special case where `cp` attempted to overwrite a directory, a lot of work needs to be done. First, we find the owner of the directory by calling the `findOwner` subroutine. If we do not find the owner, then we have to assume that it is either a vital system directory that does not really belong to a specific package, or a user created directory. In this case, we stop Vestigium and tell the user to pass proper judgment on what to do.[30] If the owner is found, we proceed to move the directory to the owner's duplicates directory, making sure to do this without using any of the wrappers. We make sure to record that directory in both the owner's `dupe.log` – so that the `removepkg` subroutine can function correctly – and the `dupe.log` of the program that is being installed. We also record the directory, its contents, and all of the other files that got a chance to be recorded in `inst.log` before the failure. Finally, a `cp` command – not through the wrappers – is run, finishing what was supposed to be accomplished. The `cp` wrapper proceeds to record the remainder of the output of the original `cp` command. Once it is finished, the wrapper proceeds to record all files that were installed in the `inst.log`

[30]For example, move the directory out manually.

of the package. The files are added to the log file with their full paths, making it easy to know where they are when parsing the log. For all duplicate files, the owner is searched for by using the `findOwner` subroutine. Note that `findOwner` does not return an owner who already has the file in its duplicates directory. This assures the ability for the duplicate file handling not to be limited to two packages only, but to apply to as many as needed. If an appropriate owner is not found, we notify the user of the problem and leave the duplicate file as is because it may be a user created file. If we do find an owner, we move the duplicate file over to the owner's duplicates directory. Finally, we log the file in the `dupe.log`s of both the program to where the duplicate was moved under and the program that is being installed.

The `ln` command creates links from a location to another file. The `ln` wrapper is line-for-line identical to the `cp` wrapper with the exception of two cases. The first is that it does not have trouble with the case where the current file is a directory. It places the symbolic link into that directory as would happen in any case. The second is that the `ln` command's output does not always give the full path to where the link is located. This is normal because a symbolic link[31] simply contains a pointer to the location of the file. That pointer does not have to be the full path. Therefore, to log correctly any created symbolic links, the `ln` wrapper checks if the output shows a beginning `/`. If this is not the case, the wrapper uses the current directory as the reference point to the link.

The `mkdir` wrapper uses the same principles applied by all of the other wrappers. The `mkdir` command, however, does not have the backup feature that the other commands have. On a positive note, we do not need to do any crazy duplicate handling with `mkdir`. All we need to do is log that it was created, or that an attempt was made to create it. The wrapper simply reads the output of mkdir and records the directories that it attempted to create in the `inst.log` of the package. Recall that the `removepkg` subroutine removes a directory only if empty. There is therefore no need to worry about needing to move a directory out

[31] Symbolic links are used more often than hard links to accommodate those users who might have separate partitions for folders.

for duplicate file handling. In fact, it could be dangerous to do so because we might move out contents with files that do not belong to the program under whose duplicates directory the directory will be stored.

3.4 Strengths and Weaknesses of Implementation

We have finished describing the full implementation of Vestigium. It is now time to step back and evaluate the strengths and weaknesses behind the implementation. After this evaluation we discuss the issues encountered when writing Vestigium and lessons learned from the experience.

The strengths of Vestigium are plenty. Even when we consider the fact that a number of features are still missing, Vestigium does a lot of things. We need not mention all of them as the reader can look through this chapter for such details, but we might as well summarize the major actions that Vestigium performs. First of all, Vestigium allows the installation of either source based or binary packages. It allows the user to switch between the installation modes with single options on the CLI. Vestigium is able to do all of the downloading, unpacking, file tracking, and dependency tracking of packages. Additionally, Vestigium provides a quick and easy way for users to synchronize with Vestigium's ports repository. All of this allows Vestigium to intelligently update systems with a single command. If we think about it, Vestigium does more than the combination of dpkg and apt-get. Most remarkably, Vestigium does all of this with a fairly small code base. Currently the code base stands at roughly three thousand lines. After the implementation of most of the missing features, Vestigium should grow to roughly five thousand lines.[32] Vestigium is relatively portable. Due to the fact that is it written in Perl, it will work on any system that Perl runs on. Binary packages can be created for any number of architectures and used with Vestigium. Source-based installation on different architectures can also be accommodated,

[32]This number is based on counting the number of missing subroutines, and estimating how many lines each might require.

provided that GNU user space can run on the architecture. Another strength of Vestigium is its ability to compile source packages with custom optimizations. Not only does Vestigium provide preset optimization flags for sub-architectures – for users who want to trust a higher power for optimizations choices – it provides the ability for users to set flags globally or on a per-package basis. A major advantage of Vestigium is the flexibility provided by the ports scripts that allow packages to be installed straight from the directory in which the package was built. Most source-based package managers build programs in a temporary directory, install the files to another temporary directory, and finally copy the installed files to the real system. Vestigium bypasses such an inefficiency without losing the ability to create binary packages of the programs.

Vestigium's greatest weakness is its current ports implementation. In particular, the script files that comprise a port are the cause of most of the trouble. The weakness is so great that Vestigium can comfortably be thought of as a useless package manager. It is not a matter of bad code in Vestigium that creates the weakness, it is simply the organization of the separate scripts and how they are intended to be used. The author discovered the weakness when attempting to create ports for packages. First of all, it is not convenient for a port writer to jump around a number of separate files to add commands. There is a reason why all information is in a single ebuild file in Portage. It helps a port writer see the big picture of what is happening and what he is working with. The author found it needlessly challenging to attempt to recall the commands that he wrote in a preceding script. The author ended up creating a Vesport utility to alleviate the inconvenience, but it was simply a work-around to a more fundamental problem.[33] Second of all – and perhaps worse of all – Vestigium's ports were designed not to have the complexity of something similar to portage's eclass. Although the complexity of the ability to use high-level functions would have been two steps backward, it would have been thousands of steps forward in the long run. Due to the fact that Vestigium does not have anything close to an eclass it

[33] Vesport turned into a very rigid text editor.

becomes exceedingly painful to re-write statements that could have a high-level functional equivalent. Another advantage to an eclass, is that you can write different functions of the same name, but that work for different architectures. What good is Vestigium's source-code portability if it does not translate into the portability of the ports?

Finally, Vestigium's ports require the modification of packages' source code. Recall, RPP, the package manager that Red Hat abandoned because the package manager mandated modifications to packages' source code. It is clearly unacceptable for a package manager today to mandate modification of source code. The reason Vestigium mandated modification was due to the "magic" behind bumping the paths of the wrappers above the normal `mv`, `cp`, `install`, `mkdir`, and `ln` commands. During the configure stage of a package's installation, the configure script would hard-wire paths to the five commands to what it found in the PATH at the time. However, when it would come time to track which files were written to the file system and where,[34] the packages' `make` scripts refused to use the wrappers. The reason for this was that the configure scripts generated `make` files with the hard-wired paths that they found before. The author attempted two work-arounds. The first was to install files to another temporary directory, and use the wrappers to copy the files from the new temporary directory to the real system like other package managers. This work-around was not satisfactory because it was taking away from Vestigium's feature of not using such temporary directories. The second work-around required changing the hard-wired paths in the `make` scripts by using sed or Perl regular expression substitutions. This was not too bad, until the author discovered that packages can have a tremendous number of `make` scripts in multiple nested directories, all requiring modifications for the hard-wired paths.[35] Make scripts can be written differently, and a single sed substitution might not work for all cases. It, therefore, becomes a pain to add ports to Vestigium in general. If it is painful to create ports, then it makes no sense to use the manager because a user keeps a package manager for software installation and updating needs.

[34] This is where the wrappers' path would get precedence in PATH.

[35] A find command can alleviate this problem by recursively going through all directories.

And so, we did not give birth to a distribution. Had Vestigium been an excellent platform for creating ports, we might have created a large enough collection to warrant the foundation of a distribution. GNU/Linux distributions need to offer something to users that might not be offered by other distributions. In our case, Vestigium is what we would offer, but what does a package manager offer if it does not have many packages due to a bad ports implementation?

We have learned how to go about developing a package manager. Most importantly, we have learned the major pitfalls a package manager developer can fall into when organizing the functions of a package manager. If the author were to start over, he would create a package manager that would parse portage's ebuilds. Not only would this spare the trouble of organizing an efficient ports systems, but it would allow users to take advantage of over eleven thousand ebuilds that can be used to build packages.

Chapter 4

Maintaining a Distribution

Once we have a distribution constructed, we have to maintain it. The bulk of the maintenance centers around keeping packages up to date. Updated packages often result in the need to modify other things, so all non-package management changes in a distribution are actually more related to package management than they may seem.

The first question a distribution might ask is how many people it needs to maintain all of the packages. Some distributions have less than a dozen developers,[1] many have less than fifty, while a number of them have more than one hundred and are always looking for more.[2] The really small distributions appear to actively maintain roughly four hundred packages, relying a lot on contributions from users.[3] The slightly larger ones maintain around one thousand or five thousand packages. For example, Gobolinux maintains an estimated nine hundred eighty one,[4] while Source Mage maintains almost five thousand.[5] The largest distributions, as we have mentioned earlier, maintain over ten thousand packages.

The small distributions,[6] appear to rely on a development model where they maintain packages as a group. In other words, people take on tasks when there is a need. The larger

[1] Crux.
[2] Gentoo, Debian, etc.
[3] You can see an example of this here: http://crux.nu/Main/About.
[4] The estimate was generated from here: http://www.gobolinux.org/recipe-store/.
[5] The estimate was generated from here: http://www.sourcemage.org/codex.
[6] Those with less than fifty developers.

distributions however have fairly strict assigned responsibilities.[7] These strict assigned responsibilities are in the form of the packages that an individual or group of individuals is responsible for maintaining. In Debian, teams of developers may collectively maintain a number of packages or individuals may maintain a number of packages. More often than not, the teams maintain more packages than individuals. Some individuals maintain more packages than a lot of teams. Furthermore, some individuals can be on numerous teams, while having their own packages to maintain.[8] Gentoo has a similar development model, where individuals maintain their own packages, while being in teams[9] that work together to maintain a number of packages.

Usually the more important packages are maintained by a team. For example, the gcc package is maintained by a team in Gentoo and Debian, called "toolchain" and "Debian GCC Maintainers" respectively. In Gentoo, the firefox package is maintained by an individual as well as a herd. Debian has a similar scheme. There are probably two reasons for this. The first is that the important packages are complex or large to the degree where a single mind cannot do all of the work to maintain them. The second is that should anything occur with an individual developer,[10] there are others who are familiar enough with his work to take on his duties. Every now and then, bugs submitted to Gentoo's bug tracking system go unfixed because of a maintainer's absence, even if users have submitted functional fixes.[11] Such an occurrence is unlikely to happen with packages that are maintained by a group since the absence of all its members at the same time is unlikely to occur.

The question still remains: How many developers does a distribution need? This question can be broken down further: How many developers are needed to maintain x number of packages? How many developers are needed to maintain x number of packages if the

[7] There is usually no centralized authority that assigns responsibilities, nor is there, in general, a "police" that enforces that people complete their responsibilities. On the contrary, developers assign responsibilities to themselves and make it up to themselves to complete the responsibilities.

[8] These facts can be observed here: http://www.us.debian.org/devel/people.

[9] Called herds.

[10] He might go on a vacation.

[11] The maintainer is needed to submit the fixes to the package management repository.

developers are expected to work y amount of time? How many packages can z developers handle? Assuming that there is no alternative to the maintainers scheme, we can use simulation to attempt to answer these questions. The use of simulation is needed because the packages need to be worked on at fairly random times. New software in the open source world is usually not released on set schedules. It can be argued that we can utilize various forecasting techniques to predict when package x might have a new release, but it is difficult to apply the same forecasting techniques to every package.[12] Data is also fairly difficult to collect from the large number of packages. Besides when new releases of packages occur, package maintenance might need to be done in response to bug reports. Forecasting occurrences of bug reports is likewise very tough, especially considering that it is impossible to even guess how many problems a package might have between releases.

4.1 Using Simulation to Estimate Personnel Requirements

Simulation allows us to construct a simplified model of a complicated system. The simplifications occur when we assume that events in the system occur randomly. Further assumptions take place to determine the random probability distributions that should be used to produce realistic results. The author has constructed a simulation of maintainers maintaining a package management repository. This simulation uses results from Gentoo's portage to construct as realistic as possible random distributions.

The first step to creating a simulation is collecting data that will determine what random distributions should be used. There are four pieces of data that this simulation collected. The first is the average time between version revisions in Gentoo's portage tree. Gentoo has a CVS/SVN repository[13] that is available for any one to download. The author wrote a Perl script[14] that uses regular expressions to extract and collect information about how

[12] Every package requires individual scrutiny to judge which forecasting technique to use.

[13] These are subversioning programs that allow a large number of people to coordinate on programming projects. CVS is the "Concurrent Versioning System". SVN is "Subversion".

[14] See retrieve-pkg-commit-times in appendix C.

much time passed between every major version "bump"[15] for each package in portage. The average time between version bumps is computed from that data. The second piece of data is the average time between non-major version bumps made by maintainers. The author simplified this to be "bug bumps", or the occurrences where a maintainer made a change in response to a discovered or reported bug. The third piece of information collected is the compile-time for every single package. This information was collected from an unsuccessful survey conducted on the official Gentoo developers' mailing lists.[16] Portage keeps a log of the packages that were successfully compiled and the times at which each compile started and ended. The author made a Perl script[17] that went through the file and extracted the compile times for packages built by the only developer that replied to the survey. The final piece of information was the average number of times a developer recompiles packages before committing an ebuild for a package. This question was asked in the survey.

Using the data collected, the author set three probability distributions to be used in the simulation. To simulate version bumps and bug bumps, the author used two exponential distributions with means extracted from mining Gentoo's CVS. The exponential distribution was chosen here because it is commonly used to express events that occur through time.[18] To simulate compile times, the author used the compile times extracted from the log file obtained from the developer. To simulate the number of times a package might need to compiled,[19] the author chose the gamma distribution with $\alpha = 1$, $\beta = 1$, and shifted $Gamma(\alpha, \beta)$ by one to the right. This gave the effect where the average number recompiles is two, but there is a decreasing probability that more recompiles are required. A

[15] A bump is a modification to a package. A version bump is when a package is updated to a new version.

[16] The survey attempted to gather logs from as many developers as possible in order to find out how long it takes to compile individual packages for each. The survey also attempted to collect information on how frequently the developers recompile their packages before committing an ebuild. You can see the inquiry on Gentoo's mailing list archives http://archives.gentoo.org/gentoo-dev/msg_141098.xml. There was only one responding developer.

[17] See pkg-comp-times-retrieve in appendix C.

[18] Ross, 28.

[19] The responding maintainer gave two as the average number of times he needs to recompile a package before committing it to the tree.

large number of recompiles, although possible, is very unlikely because the gamma function slopes down rapidly after it peaks at two. Putting everything together, the simulation generated random version bumps and bug bumps, randomly chose a target package from a list of all packages, and moved the package from that list to another list for the duration of the randomly generated bug bump[20]. The simulation then generated the number of recompiles a maintainer would do, and multiplied that by the fixed compile time for the package. The results displayed at the end were the average time that a developer ended up working on his packages per week. Each simulation run was one year.

The results from the simulation were not too useful. The main reason for this is that there was only one participating developer. His log file did not represent all of the packages available in the portage tree. This was a major determinant for the final results. Another reason the simulation was bad was that it did not take into account packages maintained by herds. Nevertheless, the simulation did a few things well. The reason for recording the package compile times was so that we could weigh some packages as more difficult to maintain than others. The big assumption made was that the larger a package is the more difficult and time consuming it is to maintain. In general the larger the package, the longer it takes to compile. Had the simulation been successful, it could be argued that distributions might use simulation to better allocate packages among maintainers by either getting new maintainers involved or reassigning packages.

4.2 The Maintainer Scheme: Pros and Cons

The maintainer scheme used by the large distributions has its benefits, but also its weaknesses. The largest benefit can be argued to be accountability. If you know who is responsible for what, you can easily determine which individual was culpable for any failures. The same works on the other end where the individual does not want to let a project down by

[20]Moving the package is done so that it does not get chosen again from the initial list too soon.

not completing his responsibilities. Another benefit is expertise. When a person is given a single thing to focus on, he becomes an expert at controlling that thing. We can think of this benefit as distribution of labor.[21] Along the same lines, users and other developers know whom to contact if they are having trouble with a package. Another benefit is that no single person ends up doing all of the work. It is dangerous if one person becomes an expert in everything to the point where no one else knows how to do anything because they never got the chance to be active. If something were to happen to the all-knowing person, the entire distribution would falter. The last benefit to the maintainer scheme is that maintainers are more likely to spend their time doing things that might be boring. When a person is assigned a responsibility, and the person is not allowed to take over others' responsibilities, he tends to spend more time completing his responsibility to perfection. For package management, a maintainer will have no excuse not to write up tedious documentation. Furthermore, a maintainer will have no excuse not to run a few extra tests on the package.

The greatest disadvantage to the maintainer scheme is that if a maintainer becomes absent, any needed maintenance to a package will not be done. Distributions have gotten around this by assigning groups to maintain important packages. Users, however, have different needs and a package that might not be important to many users might be important to some. If there is no maintainer to do maintenance then the users will either have to wait or make local changes to the source code. Such a scenario occurs fairly frequently in Gentoo.[22] Another big problem with the maintainer scheme is new or abandoned packages. Sometimes an interesting package comes around and some users want that package to be in their distribution's package management repository. Usually, there is no willing maintainer to adopt the package, either because all maintainers have their hands full with other packages, or they do not feel interested in the package. It can take a long time for

[21] Plato's Republic.

[22] See bugs 168177, 26326, 116584, and 151584 on http://bugs.gentoo.org. In each of these bugs users have done work that fixes the bugs, but no maintainer has been around to commit the fixes.

a package to be finally adopted.[23] There are countless instances where users might submit fixes, but have those fixes never – or only after a long time – make it to the package management repository because the maintainer is absent or the package no longer has a maintainer. Gentoo has recently been removing packages from the portage tree that are no longer maintained by anyone, have no one who wants to maintain them, and that have been lying untouched for a long time.[24]

In many ways it would make sense to have a system in place where users and developers could take care of needed package management without facing the "maintainer's responsibility" dilemma. Perhaps it would make sense to have a web interface where users can submit maintenance requests. A notification would be sent to the maintainer about requests. If the maintainer is available, he would quickly assign the request to himself. If not, then any willing developer or user should be able to assign the task to himself and complete the work required by the task. If the request would not be taken up by anyone, then users could battle back by voting for the request. All of the requests would be in queue, with those requests receiving the most votes on top of the list. A distribution's developer can at any time appease users by taking care of requests at the top of the queue. Developers could also act like users and vote for, or add requests to things they have not been able to fix.

4.3 Attracting and Keeping Developers

One thing on the minds of distributions is attracting developers. There are two types of developers that can be attracted: Those who desire to volunteer and those who desire to make money. Community driven distributions like Gentoo and Debian rely on the work of volunteers. For profit distributions like Red Hat rely on the work of paid employees. There are advantages and disadvantages to both schemes. Volunteers often do not need a lot of

[23] In the following bug report it took ten months to find a maintainer despite the fact that a user had been maintaining a functional ebuild: http://bugs.gentoo.org/show_bug.cgi?id=77857.

[24] Gentoo's treecleaners project: http://www.gentoo.org/proj/en/qa/treecleaners/.

motivation to get work done. They volunteered because they love doing the work that they do. A paid employee does not necessarily love the work that he does; he probably does it to make ends meet. It can be argued that a person's passion for their work drives them to create better products. This is a fairly weak argument if we posit that a person has more motivation to do better work if they need the work to feed themselves and their families.

The problem with volunteers is that they probably work part time, since they do other work to feed themselves. The result is that they have a limited amount of time to devote to their volunteer responsibilities. They might skip optional testing or a non-vital thing such as documentation. A paid employee on the other hand can devote his full time to testing and documenting what they do. Another problem with volunteers is that they are not obligated to stay on the development team. They can at any time say that they quit because they no longer have time for their hobby. A paid employee does not have any motivation to quit because he already has his or her source of money.[25]

Community based distributions attract developers by making it fairly easy to become one. These distributions allow people to work their way up from technically easy jobs to technically challenging ones. One can start as a documentation person, take on a few package maintenance responsibilities, and later join a team or teams. If you have interesting ideas, you can implement those ideas and see them adopted in the distribution simply by submitting the ideas to existing developers.[26]

Community distributions keep their developers by attempting to have a comfortable community atmosphere. Debian for example has many conferences and events.[27] These serve as a way for developers to come together and socialize. Community based distributions rely on a large number of communication mechanisms. Forums, IRC chat rooms, bug reporting systems, and mailing lists are some examples of communication mechanisms. All of these mechanisms enable a large number of people to coordinate in an organized fashion

[25]This is assuming that a volunteer might have no time for his hobby because he needs to devote himself to a full time job.

[26]Assuming that the ideas are good enough to gain acceptance.

[27]https://gallery.debconf.org/.

without ever meeting face-to-face or leaving their chairs. It can be argued that developers feel as if they are apart of something important. This feeling is certainly enough to keep a human being engaged.[28] Another thing keeping developers from leaving is the users. The users that make up a distribution's community further contribute to making developers feel a part of something grand. A developer is not just working for the benefit of themselves, but for the benefit of others.

4.4 Attracting and Keeping Users

Attracting users is something that distributions do not always do directly. There are not too many distributions that have marketing departments. It appears that marketing departments are exclusive to the for profit distributions.[29] It appears that Ubuntu is one of only a few not for profit distributions with marketing departments.[30] Other distributions rely on word of mouth or indirect marketing – news stories – to convince users to try a distribution. If one reads forum posts about how users got settled on a particular distribution, one will find that users tried a number of distributions before settling on one they preferred.

Distributions tend to attract different types of users. Ubuntu for example attracts users looking for a product that provides ease of installation and use. Gentoo attracts advanced users who want to customize their systems. Different distributions also attract either more individual users or more enterprises.[31] Individual users often need a distribution to provide a solid desktop environment with the ability to run a few basic server applications like a web server or ftp server. Individual users more often than not are probably looking for an experience where they can get things done without much hassle. This is probably because

[28] According Wickens, et al., 481, feedback motivates humans to take actions.

[29] The following is an opening for a marketing job at Red Hat:
http://redhat.hrdpt.com/cgi-bin/a/highlightjob.cgi?jobid=1421.
Here is a story that mentions SUSE have a marketing vice president and therefore having a marketing department:
http://www.newsfactor.com/perl/story/21799.html.

[30] https://launchpad.net/ ubuntu-marketing.

[31] Companies or organizations.

of the fact that users want to use their operating systems to relax after a long day's of work. What convinces individual users to stay with a distribution appears to be the ability to get the latest packages easily, the ability to get free support from forums or mailing lists, and the rarity of breakages.

Enterprise users value support first and foremost due to the fact that they are willing to pay for it. In an Enterprise it is very important for existing applications to remain running, even if they are out of date. Enterprises also expect that any package updates be done only to patch up security holes. So long as the distribution does not cause any breakage between updates, the enterprise will be content. However, eventually an enterprise might need packages of a certain version that cannot work on their system due to the age of other packages. For example, we can imagine the case where a new program is developed and can only work on a system that had been compiled with gcc-4. If the distribution is stuck with gcc-3, then the enterprise will have to abandon the distribution in order to run that program. This is because of the fact that a move from gcc-3 to gcc-4 will break more software than it will fix. Enterprises value stability as well as a system that allows their machines to evolve.[32]

[32]Not be deprecated.

Chapter 5

Conclusion

This thesis has examined many aspects of a typical GNU/Linux distribution. Novices may find it a helpful introduction to the concepts of GNU/Linux. It is important to realize that the package management systems we have described would not have been possible if software were closed source instead of open source. In such a case, users would have no optimization options and developers would have less concrete understanding of how applications interact with the operating system. We contend that users and developers alike should have quick and easy ways to find out which files make up the programs on their systems.

In the first chapter we introduced optimization. If we should conclude anything from our discussion of optimization, it is not that we have to use obscure flags such as `-ftracer` to attain one percent gains in performance. Rather, we should take away the principle that optimization can potentially have a significant impact on performance, and that it is often worthwhile to test combinations of flags. A package manager that allows users and developers to easily change optimization flags for software is one that is able to adapt quickly to the introduction of new technologies. The beauty of open sourced software is that it always changes and improves. When information is in the open people are able to come together to make it grow.

We have discussed how package management in GNU/Linux distributions works. We have covered most of the general uses for package management. Although we have described many of the major package managers, there are still some that we have not decribed. A more thorough treatment would take a longer time and more funding since there are some unique package managers that are available only to paying customers. For example, the Red Hat Enterprise distribution employs the up2date[1] package management frontend for RPM. The only way for a user to play around with up2date is to purchase Red Hat Enterprise – or, like the author, become employed at a company that uses Red Hat Enterprise. The technologies behind some of the package managers we described were touched upon superficially. Gentoo's portage, for example, is far more complicated than the author's description. Not much work has been done in academia on package management. Most of the information about package management is in the implementations, mailing list discussions, bug tracking systems, and documentation for users.

In the implementation of Vestigium an attempt was made to contribute package management ideas to the world. There were three main ideas. The first was to create an easy way for users to set CFLAGS and CXXFLAGS manually on a package by package basis. As shown in our benchmarking results, the effect of optimizations might differ between applications. If users can focus on individual applications, they can optimize their systems better. More importantly, developers who create packages for distributions might find it very efficient to be able to optimize packages using the features provided by the package manager itself.

The second major idea was the file collision handling system. File collisions slow down development for distributions that have a large number of developers. A package cannot be marked stable in portage until it has no file collisions. Red Hat's mailing lists are still filled with instances in which developers' packages conflict. The file collision handling system implemented in Vestigium is based on the simple concept of file rotation that enables users

[1]To be more accurate, Red Hat employed up2date until Red Hat Enterprise Linux 5.

and developers quickly to figure out what might be wrong and how to fix it.

The third major idea was seamless optimized binary and source based packaging in the same package manager. If this idea is implemented properly, users get the best of the best of package management systems. When users need to install a package quickly, they fetch the binary. When they want to customize their packages a little, they install from source. Furthermore, users are always able to install binaries that are optimized for their systems. Developers might also like this feature, because it permits them to test package installation conveniently.

One often hears the argument that open source software cannot generate money from anything other than support. This argument stems from the fact that when source code is available, anyone can simply download, compile, and run it without paying for precompiled binaries. There is, however, value not in precompiled binaries, but in the work that needs to be done to create the precompiled binaries. For example, it might make sense for a distribution to sell subscriptions to users for access to the distribution's package management repository.[2] No user who needs to get work done would want to download source code for all of the packages he needs and install them the LFS way. Users, especially enterprises, are willing to pay for an efficient package management system that can enable them to keep up-to-date, to customize their systems to a fair degree, and to request special things to be added to the package management system. None of these desires are capable of fulfillment without an open development model, a package manager that is able to compile packages from source, a package manager that requires minimal expertise, and a package manager that can perform its functions quickly and correctly.

[2]Red Hat employs this scheme.

5.1 Further Research

In chapter four, we mentioned the creation of an interface to help developers manage and prioritize package management tasks. The author submitted a proposal to Google Summer Code 2007 to create such an interface for the Gentoo Linux distribution. This proposal has been accepted. Once the interface is implemented and ready for deployment, we will see whether or not the theories about solving the limitations of the maintainer scheme are correct.

As we discussed at the end of the third chapter, Vestigium is able to accomplish some things well, but is inefficient in a number of important respects. Overall, it is not practical to attempt to transform Vestigium into a program to be used by real users. Its interface, features, nomenclature, code layout, and even the ideas behind it need to be reorganized. If we were to take the challenge of creating a package manager one more time, we would have to consider combining the best aspects of RPM and portage. Not only would this conceivably attract developers who know the ins and outs of working with existing schemes, but it would be a good way to implement the seamless source and binary based installation feature. There is far less demand among users for file collision handling and optimization customization than there is for features which allow them to seamlessly work with binary and source package installation.

If further research is to be conducted, it must be oriented toward developing a package manager that has great usability among a wide variety of user types. Casual desktop users need graphical user interface aids. Systems administrators need CLI aids and customization options. Enterprise users need speed, consistency, and adaptability. Solutions already exist that appeal to these and other user groups individually. A package manager that can encompass a very wide variety of users is one which can attain commercial success.

Bibliography

1. Bailey, Edward C. Maximum RPM. Red Hat, Inc., Durham, NC. 2000.

2. Beekmans, Gerard. Linux From Scratch: Version 6.2. Gerard Beekmans. 2006.

3. Drepper, Ulrich and Ingo Molnar. The Native POSIX Thread Library for Linux. Red Hat, Inc., Durham, NC. February 2005.

4. Jelnek, Jakub. Prelink. Red Hat, Inc., Durham, NC. March 2004.

5. Nahmias, Steven. Production and Operations Analysis. Fifth Edition. McGraw-Hill/Irwin, New York, NY. 2005.

6. Ross, Sheldon M. Simulation. Fourth Edition. Elsevier, Burlington, MA. 2006.

7. Wickens, Chistopher D., et al. An Introduction to Human Factors Engineering. Second Edition. Pearson Education, Inc., Upper Saddle River, New Jersey. 2004.

Appendix A

Understanding Levels of Optimization

There is a simple way think about levels of optimizations. When a person rolls their automobile into a gasoline station, he is provided with three choices of gasoline types: regular, unleaded, and premium. All three will run on his automobile, however, premium will make the car drive better. The difference is only slightly noticeable, but it exists. unleaded will make the car run slightly better than on regular, but slightly worse than on premium. Thus when a user installs a distribution that is optimized for i386 and the user is on a machine that can support up to i686, it will be as if the user is filling his computer with regular gasoline. Regular gasoline is cheaper to produce, just as it is easier to produce the i386 binaries that fit all systems. If the user installs an i686 optimized distribution, then his system will be slightly more responsive, just as a vehicle running premium gasoline.

A similar analogy applies when choosing binaries compiled with `-O1`, `-O2`, etc. We can posit that `-O1` is regular gasoline, `-O2` is unleaded, and `-O3` is premium. The `-O0` flag can be seen as the gasoline mixed with water and sold by scammers. With `-O2` a user's system will run smoothly, but with `-O3`, it will run a little more smoothly. The cost of premium gasoline is higher. The cost of `-O3` is higher in terms of the size of the resulting binary.

Appendix B

Circular Dependencies

Circular Dependencies is an interesting problem that high level package managers have faced – particularly the binary based ones like apt or rpm. These problems have been mostly worked out,[1] but have bugged users in the past. A circular dependency occurs when a package needs a dependency, but that dependency will not install because it has the following dependency: the package that we are attempting to install in the first place. In other words, we install package *x*. Package *x* asks for *y*. Package *y* however, asks for package *x* to be installed when we attempt to install *y*. This is simplest form of a circular dependency. More commonly, circular dependencies occur with more than two packages. We think of the simplest circular dependency as when two dogs chase each others tail, but the more common case as when a dozen dogs are chasing a distinct dog's tail.

There are certain methods to combat circular dependencies. The easiest way is to check which package has the least dependencies that it needs before it can be installed. Once that package is found, it is to be installed by applying the package manager's "force" option. After that, we repeat until all packages are installed. Another method is draw out the circle and force the installation of the set of packages that will most quickly eliminate circularity. The latter can be a lot tougher because there might be a very complicated circle or circles of dependencies.

[1] Developers have adapted to dealing with them.

Appendix C

Source Code

You can find the complete source code for sysmark, Vestigium, Vesport, and the simulation here: http://afalko.homelinux.net/thesis. At the same location you can find the results yielded by sysmark and the results yielded by the Simulation.

The following files are contained in the following pages:

1. `/source/vestigium/vestigium`: main Vestigium executable.
2. `/source/vestigium/cont.lib`: library for Vestigium.
3. `/source/vestigium/names.lib`: library for Vestigium.
4. `/source/vestigium/search.lib`: library for Vestigium.
5. `/source/vestigium/wrappers/cp`: cp wrapper for Vestigium.
6. `/source/vestigium/wrappers/install`: install wrapper for Vestigium.
7. `/source/vestigium/wrappers/ln`: ln wrapper for Vestigium.
8. `/source/vestigium/wrappers/mkdir`: mkdir wrapper for Vestigium.
9. `/source/vestigium/wrappers/mv`: mv wrapper for Vestigium.
10. `/source/vestigium/vestigium.comf`: Vestigium's configuration file.
11. `/source/vestigium/subarchs`: subarchs database.
12. `/source/vestigium/vesport`: Vestigium's vesport utility.
13. `/source/vestigium/Makefile`: Makefile for Vestigium.
14. `/source/sysmark-1.0.2/sysmark`: main sysmark executable.
15. `/source/sysmark-1.0.2/parse`: code to parse sysmark output.
16. `/source/sysmark-1.0.2/README`: sysmark readme documentation.
17. `/source/sysmark-1.0.2/Makefile`: Makefile for sysmark.
18. `/package/simulation/main`: main simulation executable.
19. `/package/simulation/maintainers-create`: collects information about which packages are assigned to whom.
20. `/package/simulation/pkg-comp-times-retrieve`: collects package compile times.
21. `/package/simulation/retrieve-pkg-commit-times`: collects average time between bug bumps and version bumps.

C.1 Vestigium

C.1.1 vestigium

```
#!/usr/bin/perl -w
# Copyright 2006 Andrey Falko
# Distributed under the terms of the GNU General Public License v2
# Vestigium alpha version (0.0.1)

use strict;
use vars qw { $debug $originalPATH $libDir $portDir $tmpDir $workDir $logDir $configFile $backupDir $downloadDir $wrapperPath
        @options @packageList @mirrors
        $arch $subarch $chost $cflags $cxxflags $ldflags
        $applydupes $backup $binaryOnly $clean $depclean
        $downloadOnly
        $flipdupes $force $listfiles $nodep $noremove
        $package $pkgtype $prompt $rebuild $remdupes $remove $search
        $searchdesc $searchfile $sync $verify
        @installQueue @updateQueue @uninstallQueue @rebuildQueue @backupQueue };

print "You gave me nothing to do on the command line.\n" if (! @ARGV);

$debug = 0;
$configFile = "/etc/vestigium.conf";
$originalPATH = "$ENV('PATH'}";
open CONFIG, $configFile
        or die "Unable to open the configuration file: $!";
#First filter out the #Comment lines.
while (<CONFIG>) {
        chomp $_;
        print "$_\n" if ($debug && ! /^#/ && !($_ eq ""));
        push @options, $_ if (! /^#/ && !($_ eq ""));
}
close CONFIG;
#Set the varibles.
SWITCH: for (@options) {
                /^libdir\s*=\s*(\S+)/           && do { $libDir = $1; next };
                /^portdir\s*=\s*(\S+)/          && do { $portDir = $1; next };
                /^logdir\s*=\s*(\S+)/           && do { $logDir = $1; next };
                /^backupdir\s*=\s*(\S+)/        && do { $backupDir = $1; next };
                /^tmpdir\s*=\s*(\S+)/           && do { $tmpDir = $1; next };
                /^downloaddir\s*=\s*(\S+)/      && do { $downloadDir = $1; next };
                /^wrapperpath\s*=\s*(\S+)/      && do { $wrapperPath = $1; next };
                /^packagetype\s*=\s*(\S+)/      && do { $pkgtype = $1; next };
                /^arch\s*=\s*(\S+)/             && do { $arch = $1; next };
                /^subarch\s*=\s*(\S+)/          && do { $subarch = $1; next };
                /^chost\s*=\s*\"(\S+)\"/        && do { $chost = $1; next };
                /^cflags\s*=\s*\"(.*)\"/        && do { $cflags = $1; next };
                /^cxxflags\s*=\s*\"(.*)\"/      && do { $cxxflags = $1; next };
                /^ldflags\s*=\s*\"(.*)\"/       && do { $ldflags = $1; next };
                /^mirror\s*=\s*(.+)/            && do { @mirrors = split /\s/, $1; next };
                /^nosync\s*=\s*(\S+)/           && do { if ($1 eq "true") { $sync = 0 } else { $sync = 1 } next };
                /^nodep\s*=\s*(\S+)/            && do { if ($1 eq "true") { $nodep = 1 } else { $nodep = 0 } next };
                /^debug\s*=\s*(\S+)/            && do { if ($1 eq "true") { $debug = 1 } else { $debug = 0 } next };
                /^prompt\s*=\s*(\S+)/           && do { if ($1 eq "true") { $prompt = 1 } else { $prompt = 0 } next };
                /^cleandeps\s*=\s*(\S+)/        && do { if ($1 eq "true") { $clean = 1 } else { $clean = 0 } next };
                /^alwaysforce\s*=\s*(\S+)/      && do { if ($1 eq "true") { $force = 1 } else { $force = 0 } next };
                /^flipdupes\s*=\s*(\S+)/        && do { if ($1 eq "true") { $flipdupes = 1 } else { $flipdupes = 0 } next };
                /^alwayssearchdeps\s*=\s*(\S+)/ && do { if ($1 eq "true") { $searchdesc = 1 } else { $searchdesc = 0 } next };
                /^createbinaries\s*=\s*(\S+)/   && do { if ($1 eq "true") { $package = 1 } else { $package = 0 } next };
                /^onlycreatebinaries\s*=\s*(\S+)/ && do { if ($1 eq "true") { $binaryOnly = 1 } else { $binaryOnly = 0 } next };
                do { warn "Warning: I did not understand $_ in $configFile." };
        }

system "rm -Rf $tmpDir/*";

if ($subarch) {
        open SUBARCHSFILE, "$libDir/subarchs"; #File containing architechture varibles
        my $start = 0;
        for (<SUBARCHSFILE>) {
                chomp;
                if (/^$subarch/) {
                        $start = 1;
                        next;
                }
                next if (! $start);
                /chost\s*=\s*\"(\S+)\"/          && do { $ENV{CHOST} = $1; next };
                /cflags\s*=\s*\"(.*)\"/          && do { $ENV{CFLAGS} = $1; next };
                /cxxflags\s*=\s*\"(.*)\"/        && do { $ENV{CXXFLAGS} = $1; next };
                /ldflags\s*=\s*\"(.*)\"/         && do { $ENV{LDFLAGS} = $1; next };
                /^end/                           && do { $start = 0; last };
                do { die "Bug: When parsing subarchsfile. \$_ = $_"};
        }
}

if ($cflags) {
        $ENV{CFLAGS} = $cflags;
        $ENV{CXXFLAGS} = $cflags;
}
if ($cxxflags) {
        $ENV{CXXFLAGS} = $cxxflags;
```

```
}

if ($chost) {
        $ENV{CHOST} = $chost;
}

if ($ldflags) {
        $ENV{LDFLAGS} = $ldflags;
}

#Make 100% certain that we do not lose the flags:
$cflags = $ENV{CFLAGS};
$cxxflags = $ENV{CXXFLAGS};
$chost = $ENV{CHOST};
$ldflags = $ENV{LDFLAGS};

use lib "/usr/lib/vestigium";

require "search.lib";
require "names.lib";
require "cont.lib";

unless ("$ENV{USER}" eq 'root') {
        print "You are not logged in as root. Installing or removing anything will most definitly fail.\n";
        &continuePrompt;
}

@options = 0; #We are done with this list for now, and should clear it for later use.
shift @options;
#Set more default varibles.
$downloadOnly = 0;
$remove = 0;
$noremove = 0;
$depclean = 1;
$remdupes = 0;
$applydupes = 0;
$search = 0;
$searchfile = 0;
$listfiles = 0;
$rebuild = 0;
$backup = 0;
mkdir $tmpDir;

for (@ARGV) {
        if (/^--?/) {
                if (! @options) {
                        push @options, $_;
                } elsif (! &checkIfDuplicate($_, \@options) ) {
                        push @options, $_;
                } else {
                        warn "Duplicate input parameter, $_, ignored";
                }
        } elsif (/everything/) {
                if (scalar @packageList) {
                        warn "There appears to be stuff in the package list; resetting package list
If you would like to add extra packages, add them after \"everything\" and do not add \"everything\" a second time";
                        @packageList = undef;
                }
                my @tempList = glob "$logDir/*";
                my @pkgGlob;
                for (@tempList) {
                        @pkgGlob = glob "$_/*";
                }
                @packageList = map { &getBasename($_) } @pkgGlob;
        } else {
                if (! @packageList) {
                        push @packageList, $_;
                } elsif (! &checkIfDuplicate($_, \@packageList) ) {
                        push @packageList, $_;
                } else {
                        warn "Duplicate input parameter, $_, ignored";
                }
        }
        if ((! scalar @packageList) && (! scalar @options)) { #If nothing was given on the command line.
                &printHelp;
        }
}

SWITCH: for (@options) { #The ^ are needed so that the regular expession matches do not match something
 #that they are not supposed to. e.g. -r is normally also going to match --remdupes.
                /--applybackup/ && do { $pkgtype = "backup"; next };
                /^-a|--applydupes/ && do {$applydupes = 1; next };
                /--backup/      && do { $backup = 1; next };
                /--binary/      && do { $binaryOnly = 1; next };
                /--buildsource/ && do { $pkgtype = "source"; next };
                /^-c|--clean/   && do { $clean = 1; next };
                /--debug/       && do { $debug = 1; next };
                /--depclean/    && do { $depclean = 1; next };
                /^-d|--download/ && do { $downloadOnly = 1; next };
                /--flipdupes/   && do { $flipdupes = 1; next };
                /^-f|--force/   && do { $force = 1; next };
                /^-h|--help/    && do { &printHelp };
```

```
                /^-i|--installbinary/ && do { $pkgtype = "binary"; next };
                /^-l|--listfiles/ && do { $listfiles = 1; next };
                /--nodep/       && do { $nodep = 1; next };
                /--noremove/    && do { $noremove = 1; next };
                /--package/     && do { $package = 1; next };
                /--prompt/      && do { $prompt = 1; next };
                /--rebuild/     && do { $rebuild = 1; next };
                /--remdupes/    && do { $remdupes = 1; next };
                /^-r|--remove/  && do { $remove = 1; next };
                /^-s|--search/  && do { $search = 1; next };
                /^-S|--searchdesc/ && do { $search = 1; $searchdesc = 1; next };
                /^-b|--searchfile/ && do { $searchfile = 1; next };
                /--sync/        && do { if (!$sync) {
                                        $sync = 1;
                                    } else {
                                        $sync = 0;
                                    }
                                    next;
                                };
                /^-v|--verify/  && do { $verify = 1; next };
                do { print "Option $_ was not understood, would you like to continue (N/y)? ";
                        next if (<STDIN> =~ /y/);
                        exit 0;
                };
        }

@options = undef; #We are done with it, and might as well clear some memory.

#Let do some obvious error checking so that
#we can prevent or warn a user before they
#trip themselves.
&error("I don't know which to do, search or remove. Please choose one of them.\n")
if (($search || $searchdesc || $searchfile || $listfiles) && $remove);
&error("I don't know which to do, rebuild or remove. Please choose one of them.\n")
if ($rebuild && $remove);
&error("Create something and remove it? Write your own package manager if you want to do this.\n")
if (($applydupes || $backup || $binaryOnly || $noremove || $package) && $remove);
&error("You can either search for a particular package or list the files of a particular package.\n")
if (($search || $searchdesc || $searchfile || $verify) && $listfiles);
&error("You can not list files and backup at the same time. Choose which to do.\n")
if ($listfiles && $backup);
&error("You can use rebuild or verify one at a time.\n")
if ($rebuild && $verify);
#And...I'll finish this later.

print "Debug: packageList contains = @packageList\n";

if ($sync) {
        &sync;
}

if ($search) {
        for (@packageList) {
                my @found = &findProg($_);
                next if (! @found);
                print "\"$_\" matched the following availible versions of programs:\n";
                my $str = join "\n", @found;
                print "$str\n";
                for my $progVer (@found) {
                        my @foundInst = &findInstalledProgVer($progVer);
                        if (! $foundInst[0]) {
                                print "$progVer is not installed\n";
                                next;
                        }
                        my $str = join "\n", @foundInst;
                        print "I found the following installed: \n$str\n";
                }
        }
        exit 0;
}

if ($remdupes) {
        for (@packageList) {
                my @installed = &findInstalledProg($_);
                if (! $installed[0]) {
                        print "No versions of $_ installed and therefore there is nothing for me to do";
                        &continuePrompt;
                }
                @installed = sort @installed;
                my $progVer = pop @installed;
                &removeDupes($progVer);
        }
        exit 0;
}

if ($listfiles) {
        for (@packageList) {
                print "Files belonging to $_ are: \n";
                &listFiles($_);
        }
        exit 0;
}
```

```
if ($verify) {
        for (@packageList) {
                print "Files belonging to $_ are: \n";
                my @files = &listFiles($_);
                my @bad;
                for (@files) {
                        chomp;
                        if ( -e $_) {
                                print "$_ exists. Good.\n";
                        } else {
                                print "$_ does not exist! Bad.\n";
                                push @bad, $_;
                        }
                }
                if ($bad[0]) {
                        my $missing = join "\n", @bad;
                        print "Verification found the following files missing:\n@bad\n";
                        print "You might want to rebuild the program.\n";
                } else {
                        print "Verification found that all files are intact.\n"
                }
        }
        exit 0;
}

for (@packageList) {
        if ($remove) {
                push @uninstallQueue, $_;
                next;
        }
        if ($backup) {
                push @backupQueue, $_;
                next;
        }
        my @found = &findProg($_);
        next if ($found[0] eq "1"); #findProg will return "1" if it does not find anything.
        @found = sort @found;
        my $latest = pop @found;
        my @installed = &findInstalledProg($_);
        if (!$installed[0]) {
                push @installQueue, $latest;
        } elsif (grep $_ eq $latest, @installed) {
                push @rebuildQueue, $latest if ($rebuild || $pkgtype eq "backup");
        } elsif (grep /$_/, @installed) {
                push @updateQueue, $latest;
        } else {
                die "Bug Alert! [Sirens]\n";
        }
}

{ # We are scoping @depQueue
my @depQueue;
for my $progVer (@installQueue) {
        my $prog = &getProgName($progVer);
        my $let = &getFirstLet($prog);
        if (! $nodep) {
                open INFO, "$portDir/$let/$prog/$progVer/INFO";
                my $deps;
                for (<INFO>) {
                        if (/^deps\s*=\s*\"(.*)\"/) {
                                $deps = $1;
                        }
                }
                my @deps;
                if ($deps) {
                        @deps = split /\n/, $deps;
                }
                for (@deps) {
                        my @installed = &findInstalledProg($_);
                        if (!$installed[0]) {
                                my @found = &findProg($_);
                                @found = sort @found;
                                my $latest = pop @found;
                                if (! &checkIfDuplicate($_, \@depQueue)) {
                                        push @depQueue, $latest;
                                }
                        }
                }
        }
}

for my $progVer (@rebuildQueue) {
        my $prog = &getProgName($progVer);
        my $let = &getFirstLet($prog);
        if (! $nodep) {
                open INFO, "$portDir/$let/$prog/$progVer/INFO";
                my $deps;
                for (<INFO>) {
                        if (/^deps\s*=\s*\"(.*)\"/) {
                                $deps = $1;
                        }
```

```
                }
                my @deps;
                if ($deps) {
                        @deps = split /\n/, $deps;
                }
                for (@deps) {
                        my @installed = &findInstalledProg($_);
                        if (!$installed[0]) {
                                my @found = &findProg($_);
                                @found = sort @found;
                                my $latest = pop @found;
                                if (! &checkIfDuplicate($_, \@depQueue)) {
                                        push @depQueue, $latest;
                                }
                        }
                }
        }
}

for my $progVer (@updateQueue) {
        my $prog = &getProgName($progVer);
        my $let = &getFirstLet($prog);
        if (! $nodep) {
                open INFO, "$portDir/$let/$prog/$progVer/INFO";
                my $deps;
                for (<INFO>) {
                        if (/^deps\s*=\s*\"(.*)\"/) {
                                $deps = $1;
                        }
                }
                my @deps;
                if ($deps) {
                        @deps = split /\n/, $deps;
                }
                for (@deps) {
                        my @installed = &findInstalledProg($_);
                        if (!$installed[0]) {
                                my @found = &findProg($_);
                                @found = sort @found;
                                my $latest = pop @found;
                                if (! &checkIfDuplicate($_, \@depQueue)) {
                                        push @depQueue, $latest;
                                }
                        }
                }
        }
}

#Recursivly find dependencies for dependencies
for my $depVer (@depQueue) {
        my $dep = &getProgName($depVer);
        my $let = &getFirstLet($dep);
        open INFO, "$portDir/$let/$dep/$depVer/INFO";
        my $deps;
        for (<INFO>) {
                if (/^deps\s*=\s*\"(.*)\"/) {
                        $deps = $1;
                }
        }
        my @deps;
        if ($deps) {
                @deps = split /\n/, $deps;
        }
        for (@deps) {
                my @installed = &findInstalledProg($_);
                if (!$installed[0]) {
                        my @found = &findProg($_);
                        @found = sort @found;
                        my $latest = pop @found;
                        if (! &checkIfDuplicate($_, \@depQueue)) {
                                push @depQueue, $latest;
                        }
                }
        }
}

for (@depQueue) {
        #Need to sort between update queue and install queue.
        my @installed = &findInstalledProg($_);
        if (@installed) {
                unshift @updateQueue, $_;
        } else {
                unshift @installQueue, $_;
        }
}
} # Scope for @depQueue finished

if ($prompt) {
        print "I am going to install ". @installQueue ." packages: \n" if (@installQueue);
        for (@installQueue) {
                print "   $_\n";
        }
```

```
        print "I am going to rebuild ". @rebuildQueue ." packages: \n" if (@rebuildQueue);
        for (@rebuildQueue) {
                print "   $_\n";
        }
        print "I am going to update ". @updateQueue ." packages: \n" if (@updateQueue);
        for (@updateQueue) {
                print "   $_\n";
        }
        print "I am going to remove ". @uninstallQueue ." packages: \n" if (@uninstallQueue);
        for (@uninstallQueue) {
                print "   $_\n";
        }
        print "I am going to backup ". @backupQueue ." packages: \n" if (@backupQueue);
        for (@backupQueue) {
                print "   $_\n";
        }
        &continuePrompt;
}

for my $progVer (@installQueue) {
        print "About to install $progVer\n" if ($debug);
        if ($pkgtype eq "source") {
                &installpkg($progVer);
        } elsif ($pkgtype eq "binary") {
                &installbin($progVer);
        } elsif ($pkgtype eq "backup") {
                &installbak($progVer);
        } else {
                die "Bug! This should never happen. Report it please.";
        }
}

for my $progVer (@rebuildQueue) {
        print "About to rebuild $progVer\n" if ($debug);
        if ($pkgtype eq "source") {
                &rebuildpkg($progVer);
        } elsif ($pkgtype eq "binary") {
                &rebuildbin($progVer);
        } elsif ($pkgtype eq "backup") {
                &installbak($progVer);
        } else {
                die "Bug! This should never happen. Report it please.";
        }
}

for my $progVer (@updateQueue) {
        print "About to update $progVer\n" if ($debug);
        if ($pkgtype eq "source") {
                &updatepkg($progVer);
        } elsif ($pkgtype eq "binary") {
                &updatebin($progVer);
        } elsif ($pkgtype eq "backup") {
                &installbak($progVer);
        } else {
                die "Bug! This should never happen. Report it please.";
        }
}

for my $prog (@uninstallQueue) {
        print "About to remove $prog\n" if ($debug);
        &removepkg($prog);
}

for my $prog (@backupQueue) {
        print "About to backup $prog\n" if ($debug);
        &backuppkg($prog);
}

#print "You entered the following options: @options\n";
#print 'You set $nodep = '.$nodep.', $debug = '.$debug."\n";
#print "You wish to install the following packages: @packageList\n";




sub prepareLogDir {
        #Sould be run only upon a new program install.
        warn "Debug: Entering prepareLogDir.\n" if ($debug);
        my $progVer = shift;
        $progVer =~ /(.+)-\d/;
        my $prog = $1;
        $prog =~ /(.)/;
        mkdir "$logDir/$1" or die "Could not make directory in $logDir: $!" if ( ! -d "$logDir/$1" );
        mkdir "$logDir/$1/$prog" or die "Could not make directory in $logDir/$1: $!" if ( ! -d "$logDir/$1/$prog" );
        mkdir "$logDir/$1/$prog/$progVer";
        warn "Debug: Leaving prepareLogDir.\n" if ($debug);
}

sub generateInfo { #This function generates a file in /tmp which is used by the modified mv, cp, etc. command to retrieve needed information.
        warn "Debug: Entering generateInfo\n" if ($debug);
        my $prog = shift;
```

```
        open PROFILE, ">>/tmp/vestigium/profile";
        select PROFILE;
        print "progname $prog\n";
        print "libdir $libDir\n";
        print "logdir $logDir\n";
        print "backupdir $backupDir\n";
        print "debug $debug\n";
        print "force $force\n";
        print "workDir $workDir\n";
        select STDOUT;
        close PROFILE;
        warn "Debug: Leaving generateInfo.\n" if ($debug);
}

sub backuppkg {
        warn "Debug: Entering backuppkg\n" if ($debug);
        my $prog = shift;
        my @files = &listFiles($prog);
        my $progVer;
        my $let;
        if (! ($prog =~ /\-\d/)) {
                my @progVers = &findInstalledProg($prog);
                @progVers = sort @progVers;
                $progVer = pop @progVers;
                $let = &getFirstLet($prog);
        } else {
                $progVer = $prog;
                $prog = &getProgName($progVer);
                $let = &getFirstLet($prog);
        }
        system "mkdir -p $backupDir/$let/$prog/$progVer/";
        system "tar --preserve-permissions --create --absolute-names -v --ignore-failed-read --file \
$backupDir/$let/$prog/$progVer/$progVer-$arch-$subarch.tar.gz --no-recursion -T @files";
        warn "Debug: Leaving backuppkg\n" if ($debug);
}

sub rebuildpkg {
        warn "Debug: Entering rebuildpkg\n" if ($debug);
        my $progVer = shift;
        my $prog = &getProgName($progVer);
        my $let = &getFirstLet($progVer);
        system "mv $logDir/$let/$prog/$progVer $tmpDir/"; #Save the logfiles elsewhere.
        $force = 1; #To overwrite all existing files without thinking that they are dupes.
        &installpkg($progVer);
        $force = 0; #Reset $force before we forget.
        open OLDLOG, "$tmpDir/$progVer/inst.log";
        my $matched;
        for my $old (<OLDLOG>) {
                chomp ($old);
                open NEWLOG, "$logDir/$let/$prog/$progVer/inst.log";
                for (<NEWLOG>) {
                        chomp;
                        print "Debug: Comparing NEW=$_ to OLD=$old\n" if ($debug);
                        if ($_ eq $old) {
                                #Do nothing. We do not want to delete this file.
                                $matched = 1;
                                last; #No need for more.
                        }
                        next;
                }
                if ($matched) {
                        $matched = 0;
                        next; #Keep going
                } else {
                        #File exists in OLDLOG, but not in NEWLOG; can be removed
                        print "Removing $old\n";
                        system "rm -rf $old";
                }

        }
        warn "Debug: Leaving rebuildpkg.\n" if ($debug);
}

sub installbak { #Installs backup of package if exists
        warn "Debug: Entering installbak" if ($debug);
        my $progVer = shift;
        my $prog = &getProgName($progVer);
        my $let = &getFirstLet($prog);
        my $output = `tar -xvpf $backupDir/$let/$prog/$progVer/$progVer-$arch-$subarch.tar.gz -C /`; # This should overwrite all files.
        my @output = split "\n", $output;
        system "rm -Rf $logDir/$let/$prog/$progVer/*"; # Delete the logs.
        open INSTLOG, ">>$logDir/$let/$prog/$progVer/inst.log";
        select INSTLOG;
        for (@output) {
                print "$_\n";
        }
        select STDOUT;
        warn "Debug: Leaving installbak" if ($debug);
}

sub installpkg {
        warn "Debug: Entering installpkg\n" if ($debug);
```

```
        my $progVer = shift;
        my $prog = &getProgName($progVer);
        my $let = &getFirstLet($prog);
        open INFO, "$portDir/$let/$prog/$progVer/INFO";
        my @info = <INFO>;
        close INFO;
        for (@info) {
                if (/^cflags\s*=\s*\"(.+)\"/) {
                        $ENV{CFLAGS} = $1;
                        $ENV{CXXFLAGS} = $1;
                }
                if (/^cxxflags\s*=\s*\"(.+)\"/) {
                        $ENV{CXXFLAGS} = $1;
                }
        }
        &extract($progVer);
        my $tempdir = glob "*";
        chdir $tempdir;
        &generateInfo($progVer);
        open BUILD, "$portDir/$let/$prog/$progVer/BUILD"; #or die "Could not open BUILD: $!";
        for (<BUILD>) {
                if (/cd\s+(.*)/) {
                        print "Changing working directory to: $1\n" if ($debug);
                        chdir $1;
                        next;
                }
                system $_;
        }
        close BUILD;
        &prepareLogDir($progVer);
        open INSTALL, "$portDir/$let/$prog/$progVer/INSTALL" or die "Could not open INSTALL: $!";
        $ENV{'PATH'} = "$wrapperPath:$originalPATH";
        for (<INSTALL>) {
                if (/^make/) { # This is for bug #22
                        open MAKEFILE, "Makefile" or open MAKEFILE, "makefile"; # If Makefile is not capitalized
                        my @makefile = <MAKEFILE>;
                        close MAKEFILE;
                        for (@makefile) {
                                if (/\=.+\/(install)/ || /\=.+\/(mv)/ || /\=.+\/(cp)/ || /\=.+\/(ln)/ || /\=.+\/(mkdir)/) {
                                                        #If any of these command have a /, then they probably have an
                                                        #alternname location specifed. We want the wrapper to be used.
                                                        #This is a fix for bug #22. The quicker solution could be to
                                                        #modify the in the INSTALL file, but this would require the user
                                                        #to be extra careful.
                                        # We shall only warn for now, and add substitution to INSTALL.
                                        my $cmd = $1;
                                        next if (/vestigium/);
                                        die "Bug: $cmd command is hard specified at $_ in makefile please file a bug.";
                                        #s/\=.+\/$cmd/= $cmd/;
                                        #$cmd =a 'perl -p -i.bak -w -e' . "\'s/\=.+\/$cmd/= $cmd/\'" . ' Makefile';
                                        #system $cmd; # What ever works :\
                                }
                        }
                        #open MAKEFILE, ">>Makefile" or open MAKEFILE, ">>makefile";
                        #select MAKEFILE;
                        #print $_ for (<MAKEFILE>);
                        #select STDOUT;
                        #close MAKEFILE;
                }
                print "Debug: Command to be run = $_" if ($debug);
                my @output = `$_`;
                for (@output) {
                        chomp;
                        print "Debug: OUTPUT = $_\n" if ($debug);
                        if (m/fail/i) {
                                die $_;
                        }
                }
        }
        close INSTALL;
        &resetPATH;
        open POSTINST, "$portDir/$let/$prog/$progVer/POSTINST"; #or die "Could not open POSTINST: $!";
        system $_ for (<POSTINST>);
        close POSTINST;
        &resetFlags;
        print "Debug: Installation appears finished.\n";
        system "rm -Rf /tmp/vestigium/$prog /tmp/vestigium/profile"; # Delete temporary directories of $prog and profile only
        &backuppkg($progVer) if ($package);
        warn "Debug: Leaving installpkg.\n" if ($debug);
}

sub removepkg {
        my $prog = shift;
        my @found = &findInstalledProg($prog);
        if (!$found[0]) {
                print "No installed versions of $prog were found.\n";
                &continuePrompt;
        } else {
                for my $progVer (@found) {
                        my $prog = &getProgName($progVer);
                        my $let = &getFirstLet($prog);
                        open INSTALLLOG, "$logDir/$let/$prog/$progVer/inst.log"
```

```
                                or warn "When removing $progVer, could not open inst.log: $!";
                        open DUPELOG, "$logDir/$let/$prog/$progVer/dupe.log"
                                if ( -e "$logDir/$let/$prog/$progVer/dupe.log" );
                        my @dupes = <DUPELOG>;
                        close DUPELOG;
                        system "rm $backupDir/$let/$prog/$progVer/dupes -Rf";     #Remove the dupes
                        for (reverse <INSTALLLOG>) {
                                chomp;
                                my $baseName = &getBasename($_);
                                if (! -e "$backupDir/$prog/$progVer/dupes/$baseName") { #Uninstall every file with the exception
                                                                                        #of those found in backupDir.
                                        if ( -d $_) {
                                                print "Removing dir $_\n";
                                                system "rmdir $_"; #removes dir if and only if empty.
                                                next;
                                        }
                                        print "Removing $_\n";
                                        system "rm $_ -f"; #Lets hope that the logs don't have a single "/"!
                                }
                        }
                        for my $file (reverse @dupes) {
                                chomp $file;
                                #First figure out if the file has owners
                                print "Debug: About to find owner of $file\n";
                                my @temp = &findDupeOwner($file);
                                my @owners;
                                for my $t (@temp) { # We must remove the current owner from the list.
                                        print "Debug: $t owns $file\n" if ($debug);
                                        push @owners, $t if (! ($t eq $progVer));
                                }
                                for my $ownerProgVer (@owners) {
                                        print "Debug: About to ressurect $file of $ownerProgVer\n" if ($debug);
                                        my $ownerProg = &getProgName($ownerProgVer);
                                        my $ownerLet = &getFirstLet($ownerProg);
                                        my $dupeInBackup = &getBasename($file);
                                        if (! -e "$backupDir/$ownerLet/$ownerProg/$ownerProgVer/dupes/$dupeInBackup") {
                                                # Backup dir for the other prog does not exist, we must
                                                # check the other ones.
                                                next;
                                        } else {
                                                system "rm $file -Rf";
                                                # Move the dupe out of backup and back onto file system because file
                                                # that forced it into backup is now uninstalled
                                                system "mv $backupDir/$ownerLet/$ownerProg/$ownerProgVer/dupes/$dupeInBackup $file -f";
                                                last; #Job is done, we can return the heck out of here.
                                        }
                                        # We are still here. This is not good and requires an investigation,
# however, let us not cease the removal process.
                                        warn "Nothing found in backups to replace the removed dupes
in the system. This may be because all packages with the dupes where removed already or something
fishy is going on. If any of your programs do not work after this, please file a bug.";
                                }
                                system "rm $file" if (! @owners); # We should remove the file anyway because it does not belong to anyone.
                        }
                        print "Debug: To be deleted = $logDir/$let/$prog/$progVer\n";
                        system "rm $logDir/$let/$prog/$progVer -Rf";
                        system "rm $backupDir/$let/$prog/$progVer -Rf" if ($backup);
                        system "rmdir $logDir/$let/$prog"; #removes dir if and only if empty.
                        system "rmdir $logDir/$let";
                        system "rmdir $backupDir/$let/$prog";
                        system "rmdir $backupDir/$let";
                }
        }
}

sub extract {
        warn "Debug: Entering extract\n" if ($debug);
        my $progVer = shift;
        my $prog = &getProgName($progVer);
        my $let = &getFirstLet($prog);
        open INFO, "$portDir/$let/$prog/$progVer/INFO" or die "Could not open package info: $!";
        my @info;
        for (<INFO>) {
                chomp;
                push @info, $_;
        }
        my $info = join "", @info;
        $info =~ m/files = \"(.*)\"d/;
        my $download = $1;
        #die "INFO: $info\n$1";
        my @download = split /\s/, $download;
        mkdir "$tmpDir/$prog";
        my @patchLocs;
        for (@download) {
                my $extract = &getBasename($_);
                $extract = "$downloadDir/$extract";
                &download($download) if (! -e $extract);
                print "Debug: Extract = $extract\n" if ($debug);
                if ( -e $extract) {
                        $extract =~ /\.tar.*$/          && do { system "tar -xf $extract -C $tmpDir/$prog" };
                        $extract =~ /\.rar$/            && do { system "rar e $extract $tmpDir/$prog" };
                        $extract =~ /\.patch$/          && do { push @patchLocs, $extract };
```

```
                }
        }
        my @loc = glob "$tmpDir/$prog/*";
        print "Folder extracted: @loc\n";
        $workDir = shift @loc; #first location for now.
        chdir "$workDir";
        for (@patchLocs) {
                &applyPatch($_);
        }
        warn "Debug: Leaving extract.\n" if ($debug);
}

sub download {
        warn "Debug: Entering download\n" if ($debug);
        my $download = shift;
        print "About to download $1\n"  if ($debug);
        my $fileName = &getBasename($1);
        chdir "$downloadDir";
        system "wget $1";
        #       or die "Failed to download $fileName from $1: $!";
        if ($downloadOnly) {
                #Check md5sums
                exit 0;
        } else {
                return 1;
        }
        warn "Debug: Leaving download.\n" if ($debug);
}

sub sync {
        warn "Debug: Entering sync" if ($debug);
        print "Debug: Mirrors availible: @mirrors" if ($debug);
        for my $mirror (@mirrors) {
                chdir "$downloadDir";
                system "wget $mirror/ports/ports.tar.gz";
                next if (! -e "ports.tar.gz");
                last;
        }
        if (! -e "ports.tar.gz") {
                warn "Could not download the latest ports";
                &continuePrompt;
        }
        system "tar -xf ports.tar.gz -C $portDir";
        system "rm -rf ports.tar.gz";
        warn "Debug: Exiting sync" if ($debug);
}


sub removeDupes {
        warn "Debug: Entering removeDupes" if ($debug);
        my $progVer = shift;
        my $prog = &getProgName($progVer);
        my $let = &getFirstLet($prog);
        system "rm -rf $backupDir/$let/$prog/$progVer/dupes/*";
        warn "Debug: Entering removeDupes" if ($debug);
}

sub resetPATH {
        warn "Debug: Entering resetPath\n" if ($debug);
        $ENV{'PATH'} = "$originalPATH";
        warn "Debug: Leaving resetPath\n" if ($debug);
}

sub resetFlags {
        #We saved these earlier.
        $ENV{'CFLAGS'} = "$cflags";
        $ENV{'CXXFLAGS'} = "$cxxflags";
        $ENV{'CHOST'} = "$chost";
        $ENV{'LDFLAGS'} = "$ldflags";
}

sub applyPatch {
        warn "Debug: Entering resetPath\n" if ($debug);
        my $patchName = shift; # Full path is here.
        my $count = 0;
        while ($count < 5) {
                my $output = `patch --verbose -p$count < $patchName`;
                print "Debug: $output\n";
                if ($output =~ /succeeded/) {
                        print "Applied $patchName\n";
                        last;
                }
                $count++;
        }
        if ($count >= 5) {
                &error("Patch failed after $count attempts.\n");
        }
        warn "Debug: Leaving resetPath\n" if ($debug);
}

sub configureProg {
        warn "Debug: Entering configureProg\n" if ($debug);
```

```
        my $prog = shift;
        warn "Debug: Leaving configureProg\n" if ($debug);
}

exec "rm -rf $tmpDir/*"; # Clean up
```

C.1.2 cont.lib

```
sub printHelp {
        print "First understand that Vestigium is currently alpha software and all acceptable options are not availible.\n";
        print "\nInstalling, reinstalling or updating a program is simple. Just do: \n";
        print "vestigium <package name(s)>\n";
        print "No special options are needed. The program has to exist in the ports directory for it be processed.\n";
        print "Removing an already installed package can be done with: \n";
        print "vestigium --remove <package name(s)> or vestigium -r <package name(s)\n";
        print "You can use vestigium to search for packages and thier status (installed or not). Do this by: \n";
        print "vestigium --search <package name(s) or vestigium -s <package name(s)\n";
        print "You might want to control whether Vestigium downloads a new copy of ports or not. You would do this by \n";
        print "  appending the --sync option. Vestigium will download a new copy of ports if auto-syncing is not enabled in \n";
        print "  /etc/vestigium.conf. Vice versa will occur if auto-syncing is disabled.\n";
        print "--prompt         : use this option to tell Vestigium to ask before doing something like installing or removing packages.\n";
        print "--help           : print out this help output.\n";
        print "--download (-d)  : download the files needed for the installation of a package.\n";
        print "--debug          : enable debugging output.\n";
        print "--listfiles (-l) : list all files belonging to the package (in installation logs and backup directory).\n";
        print "--backup         : backup the files that a package owns.\n";
        print "There are many more options availible, but are not yet implemented.\n";
        exit 0;
}

sub error { #Our subroutine for kicking the drunks out of the bar.
        $_ = shift;
        die $_;
}

sub continuePrompt {
        print "Would you like to continue (N/y)? ";
        if (<STDIN> =~ /y/) {
                return;
        } else {
                exit 0;
        }
}

1
```

C.1.3 names.lib

```
sub getProgName { #Extract the program name from progname-34, i.e. return progname
        my $progVer = shift;
        $progVer =~ /(.+)-\d/;
        return $1;
}

sub getFirstLet {
        my $prog = shift;
        $prog =~ /(.)/;
        return $1;
}

sub getBasename {
        warn "Debug: Entering getBasename.\n" if ($debug);
        my $path = shift;
        my @path = split /\//, $path;
        warn "Debug: Leaving getBasename.\n" if ($debug);
        return pop @path;
}

1
```

C.1.4 search.lib

```
use lib "/usr/lib/vestigium";
require "names.lib";
require "cont.lib";

sub searchBackupsForFile { #Returns the location of the file if found
        warn "Debug: Entering searchBackupsForFile" if ($debug);
        $file = shift;
        if ($file =~ /\//) {
                $file = &getBasename($file);
        }
        my @lets = glob "$backupDir/*";
        for my $let (@lets) {
                my @progs = glob "$let/*";
                for my $prog (@progs) {
                        my @progVers = glob "$prog/*";
                        for my $progVer (@progVers) {
                                my @dupes = glob "$progVer/dupes/*";
                                for (@dupes) {
                                        $_ = &getBasename($_);
                                        warn "Debug: $_ eq? $file";
                                        if ($_ eq $file) {
                                                warn "Debug: $file found in backups in $progVer" if ($debug);
                                                return "$progVer/$file";
                                        }
                                }
                        }
                }
        }
        return undef; # File not yet backed up.
}

sub findOwner { #Given a file, finds the owner
        warn "Debug: Entering findOwner" if ($debug);
        my $file = shift;
        warn "Debug: Looking for $file" if ($debug);
        my @lets = glob "$logDir/*";
        for my $let (@lets) {
                my @progs = glob "$let/*";
                for my $prog (@progs) {
                        my @progVers = glob "$prog/*";
                        for my $progVer (@progVers) {
                                warn "Debug: progVer = $progVer" if ($debug);
                                next if (! -e "$progVer/inst.log");
                                open INSTLOG, "$progVer/inst.log" or die "Could not open inst.log in $progVer: $!";
                                open DUPELOG, "$progVer/dupe.log";
                                my $backup = undef;
                                for (<INSTLOG>) {
                                        chomp;
                                        warn "Debug: $_ eq? $file";
                                        if ($_ eq $file) {
                                                #We have to check of the file is already backed up.
                                                # If it is, then we must continue the search.
                                                $backup = &searchBackupsForFile($file);
                                                my $tmp = &getBasename($progVer);
                                                if ($backup =~ /$tmp/) {
                                                        warn "$backup exists, stopping search of the $progVer/inst.log" if ($debug);
                                                        last;
                                                }
                                                #Looks like we found the owner of the file.
                                                warn "The owner of the file is = &getBasename($progVer)" if ($debug);
                                                return &getBasename($progVer);
                                        }
                                }
                                # We should also check the dupelog
                                if (! $backup) { #Why waste cputime if we already found that the backup exists.
                                        for (<DUPELOG>) {
                                                chomp;
                                                warn "Debug: $_ eq? $file";
                                                if ($_ eq $file) {
                                                        #We have to check of the file is already backed up. If it is,
                                                        #then we must continue the search.
                                                        my $tmp = &getBasename($progVer);
                                                        $backup = &searchBackupsForFile($file);
                                                        if ($backup =~ /$tmp/) {
                                                                warn "$backup exists, stopping search of the $progVer/dupe.log" if ($debug);
                                                                last;
                                                        }
                                                        warn "The owner of the file is = &getBasename($progVer)" if ($debug);
                                                        return &getBasename($progVer);
                                                }
                                        }
                                }
                        }
                }
        }
}

sub findDupeOwner { #Given a file, finds the owner
        warn "Debug: Entering findOwner\n" if ($debug);
        my $file = shift;
```

```
        my @owners;
        warn "Debug: Looking for $file\n" if ($debug);
        my @lets = glob "$logDir/*";
        for my $let (@lets) {
                my @progs = glob "$let/*";
                for my $prog (@progs) {
                        my @progVers = glob "$prog/*";
                        for my $progVer (@progVers) {
                                warn "Debug: progVer = $progVer" if ($debug);
                                next if (! -e "$progVer/dupe.log");
                                open DUPELOG, "$progVer/dupe.log" or die "Could not open dupe.log in $progVer: $!";
                                #next if (! <DUPELOG>);
                                for (<DUPELOG>) {
                                        chomp;
                                        #chomp $file; # This is a tough bug to debug!
                                        warn "Debug: $_ eq? $file\"" if ($debug);
                                        if ("$_" eq "$file") {
                                                #Looks like we found the owner of the dupe.
                                                warn "Debug: Going to return &getBasename($progVer)" if ($debug);
                                                push @owners, &getBasename($progVer);
                                        }
                                }
                        }
                }
        }
        return @owners;
}

sub findProg { #Search for program. Returns list of all availible versions.
        warn "Debug: Entering findProg.\n" if ($debug);
        my $prog = shift;
        if ($prog =~ /\-\d/) {
                $prog = getProgName($prog);
        }
        for my $let (glob "$portDir/*") {
                $let = &getBasename($let);
                for (glob "$portDir/$let/*") {
                        if (/\/$prog$/) {
                                print "Debug: $prog found.\n";
                                my @version = glob "$portDir/$let/$prog/*";
                                my @ret = map { &getBasename($_) } @version;
                                return @ret;
                        } elsif (/$prog/ && $search) {
                                my @matching = glob "$portDir/*/*$prog*";
                                print "@matching\n";
                                my @return = map { &getBasename($_) } @matching;
                                @matching = undef;
                                my @found;
                                print "Did not find $prog, but it looks like we have some that look simular: @return\n" if ($debug);
                                for my $found (@return) {
                                        @matching = &findProg($found); #for my $found (@return); #Some recursion :).
                                        for my $match (@matching) {
                                                push @found, $match;
                                        }
                                }
                                return @found;
                        } else {
                                #$let = shift @letters; #Search deeper
                                #next if (! $let); #Search deeper
                                #&notFound($prog);
                        }
                }
        }
        warn "Debug: Leaving findProg.\n" if ($debug);
        &notFound($prog);
}

sub findInstalledProgVer { #Simular to findProg, but searches log dir
        warn "Debug: Entering findInstalledProg.\n" if ($debug);
        my $progVer = shift;
        if (! $progVer =~ /\-\d/) {
                return findInstalledProg($progVer);
        }
        my $prog = getProgName($progVer);
        my $let = &getFirstLet($prog);
        return undef if (! (-e "$logDir/$let") || ! (-e "$logDir/$let/$prog"));
        for (glob "$logDir/$let/$prog/*") {
                if (/\/$progVer$/) {
                        my @version = glob "$logDir/$let/$prog/*";
                        my @return = map { &getBasename($_) } @version;
                        return @return;
                }
        }
        #Nothing found
        warn "Debug: Leaving findInstalledProg.\n" if ($debug);
        return undef;
}

sub findInstalledProg { #Simular to findInstalledProgVer, but returns all installed versions
        warn "Debug: Entering findInstalledProg.\n" if ($debug);
        my $prog = shift;
        if ($prog =~ /\-\d/) {
```

```
                return findInstalledProgVer($prog);
        }
        my $let = &getFirstLet($prog);
        return undef if (! (-e "$logDir/$let") || ! (-e "$logDir/$let/$prog"));
        my @version = glob "$logDir/$let/$prog/*";
        my @return = map { &getBasename($_) } @version;
        warn "Debug: Leaving findInstalledProg.\n" if ($debug);
        return @return;
}

sub listFiles {
        warn "Debug: Entering listFiles.\n" if ($debug);
        my $prog = shift;
        print "Debug: Prog = $prog\n" if ($debug);
        if (! ($prog =~ /\-\d/)) { #If prog is not have a version attached.
                my @progVers = &findInstalledProg($prog);
                print "Debug: \@progVer = @progVers\n" if ($debug);
                if (! $progVers[0]) {
                        warn "$prog not found or not installed.\n";
                        &continuePrompt;
                } else {
                        @progVers = sort @progVers;
                        my $progVer = pop @progVers;
                        my $let = &getFirstLet($prog);
                        print "Debug: Opening $logDir/$let/$prog/$progVer/inst.log\n" if($debug);
                        open INSTLOG, "$logDir/$let/$prog/$progVer/inst.log"
                                or do { print "$prog not found or installed\n"; &continuePrompt };
                        my @files;
                        while (<INSTLOG>) {
                                if ($listfiles) {
                                        print $_;
                                } else {
                                        chomp;
                                        push @files, $_;
                                }
                        }
                        for (glob "$backupDir/$let/$prog/$progVer/dupes/*") {
                                if ($listfiles) {
                                        print $_;
                                } else {
                                        chomp;
                                        push @files, $_;
                                }
                        }
                        warn "Debug: Leaving listFiles" if ($debug);
                        if ($listfiles) {
                                exit 0;
                        } else {
                                return @files;
                        }
                }
        } else {
                $progVer = $prog;
                $prog = &getProgName($progVer);
                $let = &getFirstLet($prog);
                print "Debug: Opening $logDir/$let/$prog/$progVer/inst.log\n" if($debug);
                open INSTLOG, "$logDir/$let/$prog/$progVer/inst.log";
                my @files;
                while (<INSTLOG>) {
                        if ($listfiles) {
                                print $_;
                        } else {
                                chomp;
                                push @files, $_;
                        }
                }
                warn "Debug: Leaving listFiles" if ($debug);
                if ($listfiles) {
                        exit 0;
                } else {
                        return @files;
                }
        }
}

sub notFound {
        warn "Debug: Entering notFound.\n" if ($debug);
        my $thing = shift;
        warn "$thing not found. ";
        &continuePrompt;
        warn "Debug: Leaving notFound.\n" if ($debug);
}

sub checkIfDuplicate { # Recives a value to be found in list reference. Returns 1 if found, 0 if not.
        my $value = shift;
        my $listref = shift;
        for (@$listref) {
                return 1 if ($_ eq $value);
        }
        return 0; # Not found
}
```

1

C.1.5 cp

```
#!/usr/bin/perl -w
# Copyright 2006 Andrey Falko
# cp for Vestigium, version (1.0.0)

use strict;
use vars qw { $debug $progVer $logDir $libDir $backupDir $workDir $force
        @args @dupes @inst };


open PROFILE, "/tmp/vestigium/profile";
while (<PROFILE>) {
        chomp;
        /^progname\s(.+)/       && do { $progVer = $1; next };
        /^libdir\s(.+)/         && do { $libDir = $1; next };
        /^logdir\s(.+)/         && do { $logDir = $1; next };
        /^backupdir\s(.+)/      && do { $backupDir = $1; next };
        /^debug\s(.+)/          && do { $debug = $1; next };
        /^force\s(.+)/          && do { $force = $1; next };
        /^workDir\s(.*)/        && do { $workDir = $1; next };
}
close PROFILE;

use lib "/usr/lib/vestigium";

require "search.lib";
require "names.lib";

my $prog = &getProgName($progVer);
my $let = &getFirstLet($prog);

$ENV{VERSION_CONTROL} = "simple";
$ENV{SIMPLE_BACKUP_SUFFIX} = ".dupe";

for (@ARGV) {
        /^-([A-Za-z]*)b([A-Za-z]*)|--backup/
&& do { push @args, "\-$1$2" if ($1 || $2); next }; # Just in case some
    # bum descides to
    # abuse the command line.
        /^-S|--suffix/    && do { next }; # Just in case some bum descides to abuse the command line.
        /^-([A-Za-z]*)v([A-Za-z]*)|--version/
&& do { push @args, "\-$1$2" if ($1 || $2); next }; # Just in case some
    # bum descides to
    # abuse the command line.
        /^-V|--version-control/
&& do { next }; # Just in case some bum
# descides to abuse the
# command line. The beauty
# of copy and paste (Except
# when there are tpyos).
        do { push @args, $_; next };
}

chdir "$workDir" or warn "No dir to chdir to: $!";

my $args = join " ", @args;
print "CP: $args\n" if ($debug);
my @result;
if ($force) {
        @result = `/bin/cp $args -rfv 2>&1`;
} else {
        @result = `/bin/cp $args -bv 2>&1`; # We need the error output...just in case.
}

for (@result) {
        warn "CP result = $_" if ($debug);
        if (/\(backup: \`(.+)\')\$/) {
                push @dupes, $1;
                next;
        } elsif (/ \`(.+)\'$/) {
                push @inst, $1;
                next;
        } elsif (/cannot overwrite directory \`(.+)\'/) {
                print "Entering special directory overwrite handling.\n";
                # This is a very special case and requires delicate handling and commentary:
                # The reason being that the -b options does not play nice here.
                # First, we will have to move the directory out to the backup dir. and record the dupe.
                my $owningProgVer = &findOwner($1);
                if (! $owningProgVer) {
                        # If the owner is not found, then this might be a vital directory or a user created
                        # directory. All we can do is tell the user to move the directory out manually.
                        die "There was a problem: the cp command attempted to overwrite $1, which
is a directory. The owner of this directory was not found, and it was therefore not backed up.
Obviously, we did not record this in the logs or attempt to force (even if you used the --force
option with vestigium. If the program that you are installing, updating or rebuild needs this
command to go through, please manually move $1 out of the way, and run the installation command again.";
                }
                my $owningProg = &getProgName($owningProgVer);
                my $ownLet = &getFirstLet($owningProg);
                system "/bin/mkdir -p $backupDir/$ownLet/$owningProg/$owningProgVer/dupes";
                system "/bin/mv $1 $backupDir/$ownLet/$owningProg/$owningProgVer/dupes/ -f";
```

```
                open OWNERDUPE, ">>$logDir/$ownLet/$owningProg/$owningProgVer/dupe.log";
                select OWNERDUPE;
                print "$1\n";
                select STDOUT;
                close OWNERDUPE;
                open DUPE, ">>$logDir/$let/$prog/$progVer/dupe.log";
                select DUPE;
                print "$1\n";
                close DUPE;
                open INSTALL, "$logDir/$let/$prog/$progVer/inst.log"; #The following should fix bug #9
                my @dupeDirContents;
                close INSTALL;
                while (<INSTALL>) {
                        unless (/^$1/) { # We want to keep anything that does not begin with the dir we are moving out.
                                push @dupeDirContents, $_;
                        }
                }
                system "rm $logDir/$let/$prog/$progVer/inst.log"; # Clear the log.
                open INSTALL, ">>$logDir/$let/$prog/$progVer/inst.log" or die "Could not open inst.log for write: $!";
                select INSTALL;
                for (@dupeDirContents) { # Now we rebuild the inst.log.
                        print "$_\n";
                }
                select STDOUT;
                close INSTALL;

                # Now, we have to re-run the command that failed command. We do not have any worries
                # because everything that went through ok, went through ok. We can run cp without
                # the backup option and with force. We also do not have to worry about the files that will
                # fail in the same way as they did here (They won't mess up our work ;)).
                system "/bin/cp $args -rf";
        } else {
                warn "Bug: cp did not do its job with $_";
        }
}

open INST, ">>$logDir/$let/$prog/$progVer/inst.log"; #Logdirs have allegedly been created
select INST;
for (@inst) {
        warn "Logging $_ to $logDir/$let/$prog/$progVer/inst.log" if ($debug);
        print "$_\n";
}

open DUPE, ">>$logDir/$let/$prog/$progVer/dupe.log";
select DUPE;
for my $dupe (@dupes) {
        $dupe =~ s/\.dupe$//; #Strip the extention
        my $owningProgVer = &findOwner($dupe);
        if (! $owningProgVer) {
                warn "There was a problem with backing up a file on the filesystem. I have moved it from $dupe to $dupe.dupe";
                select INST;
                warn "Logging $dupe to $logDir/$let/$prog/$progVer/inst.log" if ($debug);
                print "$dupe\n";
                next;
        } else {
                my $owningProg = &getProgName($owningProgVer);
                my $ownLet = &getFirstLet($owningProg);
                my $file = &getBasename($_);
                open OWNERDUPE, ">>$logDir/$ownLet/$owningProg/$owningProgVer/dupe.log";
                select OWNERDUPE;
                system "/bin/mkdir -p $backupDir/$ownLet/$owningProg/$owningProgVer/dupes";
                system "/bin/mv $dupe.dupe $backupDir/$ownLet/$owningProg/$owningProgVer/dupes/$file";
                warn "Logging $dupe to $logDir/$ownLet/$owningProg/$owningProgVer/dupe.log" if ($debug);
                print "$dupe\n";
                select DUPE;
                warn "Logging $dupe to $logDir/$let/$prog/$progVer/dupe.log" if ($debug);
                print "$dupe\n";
        }
}
close OWNERDUPE;
close DUPE;
close INST;
```

C.1.6 install

```
#!/usr/bin/perl
# Copyright 2006 Andrey Falko
# install for Vestigium, version (1.1.0)

open PROFILE, "/tmp/vestigium/profile";
while (<PROFILE>) {
        chomp;
        /^progname\s(.+)/       && do { $progVer = $1; next };
        /^libdir\s(.+)/         && do { $libDir = $1; next };
        /^logdir\s(.+)/         && do { $logDir = $1; next };
        /^backupdir\s(.+)/      && do { $backupDir = $1; next };
        /^debug\s(.+)/          && do { $debug = $1; next };
        /^force\s(.+)/          && do { $force = $1; next };
        /^workDir\s(.*)/        && do { $workDir = $1; next };
}
close PROFILE;

use lib "/usr/lib/vestigium";

require "search.lib";
require "names.lib";

my $prog = &getProgName($progVer);
my $let = &getFirstLet($prog);

$ENV{VERSION_CONTROL} = "simple";
$ENV{SIMPLE_BACKUP_SUFFIX} = ".dupe";

for (@ARGV) {
        /^-([A-Za-z]*)b([A-Za-z]*)|--backup/
&& do { push @args, "\-$1$2" if ($1 || $2); next }; # Just in case some
    # bum descides to
    # abuse the command line.
        /^-S|--suffix/    && do { next }; # Just in case some bum
  # descides to abuse the command line.
        /^-([A-Za-z]*)v([A-Za-z]*)|--version/
&& do { push @args, "\-$1$2" if ($1 || $2); next }; # Just in case some
    # bum descides to
    # abuse the command line.
        /^-V|--version-control/
&& do { next }; # Just in case some bum
# descides to abuse the
# command line. The beauty
# of copy and paste (Except
# when there are tpyos).
        do { push @args, $_; next };
}

chdir "$workDir" or warn "No dir to chdir to: $!";

my $args = join " ", @args;
print "INSTALL: $args\n" if ($debug);
my @result;
if ($force) {
        @result = `/usr/bin/install $args -v 2>&1`;
} else {
        @result = `/usr/bin/install $args -bv 2>&1`; # We need the error output...just in case.
}

for (@result) {
        warn "INSTALL result = $_" if ($debug);
        if (/\(backup: \`(.+)\')\)$/) {
                push @dupes, $1;
                next;
        } elsif (/ \`(.+)\'$/) {
                warn "Debug: \$1 is $1" if ($debug);
                push @inst, $1;
                next;
        } elsif (/\`(.+)\' exists but is not a directory/) {
                print "Entering special directory overwrite handling.\n";
                # This is a very special case and requires delicate handling and commentary:
                # The reason being that the -b options does not play nice here.
                # First, we will have to move the directory out to the backup dir. and record the dupe.
                my $owningProgVer = &findOwner($1);
                if (! $owningProgVer) {
                        # If the owner is not found, then this might be a vital directory or a user created
                        # directory. All we can do is tell the user to move the directory out manually.
                        die "There was a problem: the install command attempted to overwrite $1,
which is a directory. The owner of this directory was not found, and it was therefore not backed
up. Obviously, we did not record this in the logs or attempt to force (even if you used the
--force option with vestigium. If the program that you are installing, updating or rebuild needs
this command to go through, please manually move $1 out of the way, and run the installation command again.";
                }
                my $owningProg = &getProgName($owningProgVer);
                my $ownLet = &getFirstLet($owningProg);
                system "/bin/mkdir -p $backupDir/$ownLet/$owningProg/$owningProgVer/dupes";
                system "/bin/mv $1 $backupDir/$ownLet/$owningProg/$owningProgVer/dupes/ -f";
                open OWNERDUPE, ">>$logDir/$ownLet/$owningProg/$owningProgVer/dupe.log";
                select OWNERDUPE;
                print "$1\n";
```

```
                select STDOUT;
                close OWNERDUPE;
                open DUPE, ">>$logDir/$let/$prog/$progVer/dupe.log";
                select DUPE;
                print "$1\n";
                close DUPE;

                open INSTALL, "$logDir/$let/$prog/$progVer/inst.log"; #The following should fix bug #9
                my @dupeDirContents;
                close INSTALL;
                while (<INSTALL>) {
                        unless (/^$1/) { # We want to keep anything that does not begin with the dir we are moving out.
                                push @dupeDirContents, $_;
                        }
                }
                system "rm $logDir/$let/$prog/$progVer/inst.log"; # Clear the log.
                open INSTALL, ">>$logDir/$let/$prog/$progVer/inst.log" or die "Could not open inst.log for write: $!";
                select INSTALL;
                for (@dupeDirContents) { # Now we rebuild the inst.log.
                        print "$_\n";
                }
                select STDOUT;
                close INSTALL;

                # Now, we have to re-run the command that failed command. We do not have any worries
                # because everything that went through ok, went through ok. We can run mv without
                # the backup option and with force. We also do not have to worry about the files that will
                # fail in the same way as they did here (They won't mess up our work ;)).
                system "/usr/bin/install $args";
        } else {
                 warn "Bug: install did not do its job with $_";
        }
}

open INST, ">>$logDir/$let/$prog/$progVer/inst.log"; #Logdirs have allegedly been created
select INST;
for (@inst) {
        warn "Logging $_ to $logDir/$let/$prog/$progVer/inst.log" if ($debug);
        print "$_\n";
}

open DUPE, ">>$logDir/$let/$prog/$progVer/dupe.log";
select DUPE;
for my $dupe (@dupes) {
        $dupe =~ s/\.dupe$//; #Strip the extention
        my $owningProgVer = &findOwner($dupe);
        if (! $owningProgVer) {
                warn "There was a problem with backing up a file on the filesystem.
I have moved it from $dupe to $dupe.dupe.";
                select INST;
                warn "Logging $dupe to $logDir/$let/$prog/$progVer/inst.log" if ($debug);
                print "$dupe\n";
                next;
        } else {
                my $owningProg = &getProgName($owningProgVer);
                my $ownLet = &getFirstLet($owningProg);
                my $file = &getBasename($dupe);
                open OWNERDUPE, ">>$logDir/$ownLet/$owningProg/$owningProgVer/dupe.log";
                select OWNERDUPE;
                system "/bin/mkdir -p $backupDir/$ownLet/$owningProg/$owningProgVer/dupes";
                system "/bin/mv $dupe.dupe $backupDir/$ownLet/$owningProg/$owningProgVer/dupes/$file";
                warn "Logging $dupe to $logDir/$ownLet/$owningProg/$owningProgVer/dupe.log" if ($debug);
                print "$dupe\n";
                select DUPE;
                warn "Logging $dupe to $logDir/$let/$prog/$progVer/dupe.log" if ($debug);
                print "$dupe\n";
        }
}
close OWNERDUPE;
close DUPE;
close INST;
```

C.1.7 ln

```
#!/usr/bin/perl -w
# Copyright 2006 Andrey Falko
# ln for Vestigium, version (1.0.0)

use strict;
use vars qw { $debug $progVer $logDir $libDir $backupDir $workDir $force
        @args @dupes @inst };


open PROFILE, "/tmp/vestigium/profile";
while (<PROFILE>) {
        chomp;
        /^progname\s(.+)/       && do { $progVer = $1; next };
        /^libdir\s(.+)/         && do { $libDir = $1; next };
        /^logdir\s(.+)/         && do { $logDir = $1; next };
        /^backupdir\s(.+)/      && do { $backupDir = $1; next };
        /^debug\s(.+)/          && do { $debug = $1; next };
        /^force\s(.+)/          && do { $force = $1; next };
        /^workDir\s(.*)/        && do { $workDir = $1; next };
}
close PROFILE;

use lib "/usr/lib/vestigium";

require "search.lib";
require "names.lib";

my $prog = &getProgName($progVer);
my $let = &getFirstLet($prog);

$ENV{VERSION_CONTROL} = "simple";
$ENV{SIMPLE_BACKUP_SUFFIX} = ".dupe";

for (@ARGV) {
        /^-([A-Za-z]*)b([A-Za-z]*)|--backup/
&& do { push @args, "\-$1$2" if ($1 || $2); next }; # Just in case some
    # bum descides to
    # abuse the command line.
        /^-S|--suffix/
&& do { next }; # Just in case some bum descides to abuse the command line.
        /^-([A-Za-z]*)v([A-Za-z]*)|--version/
&& do { push @args, "\-$1$2" if ($1 || $2); next }; # Just in case some
    # bum descides to
    # abuse the command line.
        /^-V|--version-control/
&& do { next }; # Just in case some bum descides to
        # abuse the command line. The beauty
# of copy and paste (Except when there
# are tpyos).
        do { push @args, $_; next };
}

chdir "$workDir" or warn "No dir to chdir to: $!";

my $args = join " ", @args;
print "LN: $args\n" if ($debug);
my @result;
if ($force) {
        @result = `/bin/ln $args -v 2>&1`;
} else {
        @result = `/bin/ln $args -bv 2>&1`; # We need the error output...just in case.
}

for (@result) {
        warn "LN result = $_" if ($debug);
        if (/\(backup: \`(.+)\')\)$/) {
                my $loc = $1;
                $loc = "$ENV{PWD}/$loc" if (! /^\//); # If not begining /, then we must add the current location. (Bug #20)
                push @dupes, $loc;
                next;
        } elsif (/ \`(.+)\' to/) {
                my $loc = $1;
                $loc = "$ENV{PWD}/$loc" if (! /^\//); # See comment above.
                push @inst, $loc;
                next;
        } else {
                warn "Bug: ln did not do its job with $_";
        }
}

open INST, ">>$logDir/$let/$prog/$progVer/inst.log"; #Logdirs have allegedly been created
select INST;
for (@inst) {
        warn "Logging $_ to $logDir/$let/$prog/$progVer/inst.log" if ($debug);
        print "$_\n";
}

open DUPE, ">>$logDir/$let/$prog/$progVer/dupe.log";
select DUPE;
for my $dupe (@dupes) {
```

```
        $dupe =~ s/\.dupe$//; #Strip the extention
        my $owningProgVer = &findOwner($dupe);
        if (! $owningProgVer) {
                warn "There was a problem with backing up a file on the filesystem. I have moved it from $dupe to $dupe.dupe.";
                select INST;
                warn "Logging $dupe to $logDir/$let/$prog/$progVer/inst.log" if ($debug);
                print "$dupe\n";
                next;
        } else {
                my $owningProg = &getProgName($owningProgVer);
                my $ownLet = &getFirstLet($owningProg);
                my $file = &getBasename($dupe);
                open OWNERDUPE, ">>$logDir/$ownLet/$owningProg/$owningProgVer/dupe.log";
                select OWNERDUPE;
                system "/bin/mkdir -p $backupDir/$ownLet/$owningProg/$owningProgVer/dupes";
                system "/bin/mv $dupe.dupe $backupDir/$ownLet/$owningProg/$owningProgVer/dupes/$file -f";
                warn "Logging $dupe to $logDir/$ownLet/$owningProg/$owningProgVer/dupe.log" if ($debug);
                print "$dupe\n";
                select DUPE;
                warn "Logging $dupe to $logDir/$let/$prog/$progVer/dupe.log" if ($debug);
                print "$dupe\n";
        }
}
close OWNERDUPE;
close DUPE;
close INST;
```

C.1.8 mkdir

```
#!/usr/bin/perl -w
# Copyright 2006 Andrey Falko
# mkdir for Vestigium, version (1.0.0)

use strict;
use vars qw { $debug $progVer $logDir $libDir $backupDir $workDir $force
        @args @dupes @inst };

open PROFILE, "/tmp/vestigium/profile";
while (<PROFILE>) {
        chomp;
        /^progname\s(.+)/       && do { $progVer = $1; next };
        /^libdir\s(.+)/         && do { $libDir = $1; next };
        /^logdir\s(.+)/         && do { $logDir = $1; next };
        /^backupdir\s(.+)/      && do { $backupDir = $1; next };
        /^debug\s(.+)/          && do { $debug = $1; next };
        /^force\s(.+)/          && do { $force = $1; next };
        /^workDir\s(.*)/        && do { $workDir = $1; next };
}
close PROFILE;

use lib "/usr/lib/vestigium";

require "search.lib";
require "names.lib";

my $prog = &getProgName($progVer);
my $let = &getFirstLet($prog);

for (@ARGV) {
        /^-([A-Za-z]*)v([A-Za-z]*)|--version/
&& do { push @args, "\-$1$2" if ($1 || $2); next }; # Just in case some
    # bum descides to abuse
    # the command line.
        do { push @args, $_; next };
}

chdir "$workDir" or warn "No dir to chdir to: $!";

my $args = join " ", @args;
print "MKDIR: $args\n" if ($debug);
my @result = `/bin/mkdir $args -v 2>&1`; # mkdir developers make verbose output go to STDERR. Weird huh?

for (@result) {
        print "Debug: MKDIR, result = $_\n" if ($debug);
        if (/\`(.+)\': File exists$/) {
                push @dupes, $1;
                next;
        } elsif (/ \`(.+)\'$/) {
                push @inst, $1;
                next;
        } else {
                warn "mkdir appears to have not done anything. This is normal in most circumstances." if ($debug);
                push @dupes, $1;
                next;
        }
}

open INST, ">>$logDir/$let/$prog/$progVer/inst.log"; #Logdirs have allegedly been created
select INST;
for (@inst) {
        print "$_\n";
}

# If a dir already exists, then we add it to the inst.log.
# When a package is removed, we only remove the dir if it
# empty.
for (@dupes) {
        my $dupe = $_;
        print "$_\n";
}

close INST;
```

C.1.9 mv

```
#!/usr/bin/perl -w
# Copyright 2006 Andrey Falko
# mv for Vestigium, version (1.1.0)

use strict;
use vars qw { $debug $progVer $logDir $libDir $backupDir $workDir
        @args @dupes @inst };


open PROFILE, "/tmp/vestigium/profile";
while (<PROFILE>) {
        chomp;
        /^progname\s(.+)/       && do { $progVer = $1; next };
        /^libdir\s(.+)/         && do { $libDir = $1; next };
        /^logdir\s(.+)/         && do { $logDir = $1; next };
        /^backupdir\s(.+)/      && do { $backupDir = $1; next };
        /^debug\s(.+)/          && do { $debug = $1; next };
        /^workDir\s(.*)/        && do { $workDir = $1; next };
}
close PROFILE;

use lib "/usr/lib/vestigium";

require "search.lib";
require "names.lib";

my $prog = &getProgName($progVer);
my $let = &getFirstLet($prog);

$ENV{VERSION_CONTROL} = "simple";
$ENV{SIMPLE_BACKUP_SUFFIX} = ".dupe";

for (@ARGV) {
        /^-([A-Za-z]*)b([A-Za-z]*)|--backup/
&& do { push @args, "\-$1$2" if ($1 || $2); next }; # Just in case some
    # bum descides to
    # abuse the command line.
        /^-S|--suffix/    && do { next }; # Just in case some bum
  # decides to abuse the command line.
        /^-([A-Za-z]*)v([A-Za-z]*)|--version/
&& do { push @args, "\-$1$2" if ($1 || $2); next }; # Just in case some
    # bum descides to
    # abuse the command line.
        /^-V|--version-control/ && do { next }; # Just in case some bum descides to
                                        # abuse the command line. The beauty
# of copy and paste (Except when there
# are tpyos).
        do { push @args, $_; next };
}

chdir "$workDir" or warn "No dir to chdir to: $!";

my $args = join " ", @args;
print "MV: $args\n" if ($debug);
my @result = `/bin/mv $args -bv`; # We need the error output...just in case.

for (@result) {
        warn "MV result = $_" if ($debug);
        if (/\(backup: \`(.+)\')\)$/) {
                push @dupes, $1;
                next;
        } elsif (/removed/) {
                #This is part of the verbose output, but can be ignored.
                next;
        } elsif (/ \`(.+)\'$/) {
                warn "Debug: \$1 is $1" if ($debug);
                push @inst, $1;
                next;
        } elsif (/cannot overwrite directory \`(.+)\'/) {
                print "Entering special directory overwrite handling.\n";
                # This is a very special case and requires delicate handling and commentary:
                # The reason being that the -b options does not play nice here.
                # First, we will have to move the directory out to the backup dir. and record the dupe.
                my $owningProgVer = &findOwner($1);
                if (! $owningProgVer) {
                        # If the owner is not found, then this might be a vital directory or a user created
                        # directory. All we can do is tell the user to move the directory out manually.
                        die "There was a problem: the mv command attempted to overwrite $1,
which is a directory. The owner of this directory was not found, and it was therefore not
backed up. Obviously, we did not record this in the logs or attempt to force (even if you
used the --force option with vestigium. If the program that you are installing, updating or
rebuild needs this command to go through, please manually move $1 out of the way, and run the
installation command again.";
                }
                my $owningProg = &getProgName($owningProgVer);
                my $ownLet = &getFirstLet($owningProg);
                system "/bin/mkdir -p $backupDir/$ownLet/$owningProg/$owningProgVer/dupes";
                system "/bin/mv $1 $backupDir/$ownLet/$owningProg/$owningProgVer/dupes/ -f";
                open OWNERDUPE, ">>$logDir/$ownLet/$owningProg/$owningProgVer/dupe.log";
                select OWNERDUPE;
```

```
                print "$1\n";
                select STDOUT;
                close OWNERDUPE;
                open DUPE, ">>$logDir/$let/$prog/$progVer/dupe.log";
                select DUPE;
                print "$1\n";
                close DUPE;

                open INSTALL, "$logDir/$let/$prog/$progVer/inst.log"; #The following should fix bug #9
                my @dupeDirContents;
                close INSTALL;
                while (<INSTALL>) {
                        unless (/^$1/) { # We want to keep anything that does not begin with the dir we are moving out.
                                push @dupeDirContents, $_;
                        }
                }
                system "rm $logDir/$let/$prog/$progVer/inst.log"; # Clear the log.
                open INSTALL, ">>$logDir/$let/$prog/$progVer/inst.log" or die "Could not open inst.log for write: $!";
                select INSTALL;
                for (@dupeDirContents) { # Now we rebuild the inst.log.
                        print "$_\n";
                }
                select STDOUT;
                close INSTALL;

                # Now, we have to re-run the command that failed command. We do not have any worries
                # because everything that went through ok, went through ok. We can run mv without
                # the backup option and with force. We also do not have to worry about the files that will
                # fail in the same way as they did here (They won't mess up our work ;)).
                system "/bin/mv $args -rf";
        } else {
                warn "Bug: mv did not do its job with $_";
        }
}

open INST, ">>$logDir/$let/$prog/$progVer/inst.log"; #Logdirs have allegedly been created
select INST;
for (@inst) {
        warn "Logging $_ to $logDir/$let/$prog/$progVer/inst.log" if ($debug);
        print "$_\n";
}

open DUPE, ">>$logDir/$let/$prog/$progVer/dupe.log";
select DUPE;
for my $dupe (@dupes) {
        $dupe =~ s/\.dupe$//; #Strip the extention
        my $owningProgVer = &findOwner($dupe);
        if (! $owningProgVer) {
                warn "There was a problem with backing up a file on the filesystem. I have moved it from $dupe to $dupe.dupe.";
                select INST;
                warn "Logging $dupe to $logDir/$let/$prog/$progVer/inst.log" if ($debug);
                print "$dupe\n";
                next;
        } else {
                my $owningProg = &getProgName($owningProgVer);
                my $ownLet = &getFirstLet($owningProg);
                my $file = &getBasename($dupe);
                open OWNERDUPE, ">>$logDir/$ownLet/$owningProg/$owningProgVer/dupe.log";
                select OWNERDUPE;
                system "/bin/mkdir -p $backupDir/$ownLet/$owningProg/$owningProgVer/dupes";
                system "/bin/mv $dupe.dupe $backupDir/$ownLet/$owningProg/$owningProgVer/dupes/$file";
                warn "Logging $dupe to $logDir/$ownLet/$owningProg/$owningProgVer/dupe.log" if ($debug);
                print "$dupe\n";
                select DUPE;
                warn "Logging $dupe to $logDir/$let/$prog/$progVer/dupe.log" if ($debug);
                print "$dupe\n";
        }
}
close OWNERDUPE;
close DUPE;
close INST;
```

C.1.10 vestigium.conf

```
libdir = /usr/lib/vestigium

portdir = /usr/vestigium/ports

#The main directory where logs are stored.
#Do not change this unless you know what
#you doing. If Vestigium is given a directory
#where the logs to no exist, it will assume
#nothing has ever been installed, and will
#not be able to handle the many file
#collisions that will result and it attempts
#to rebuild the system.
logdir = /var/log/vestigium

backupdir = /var/spool/vestigium

tmpdir = /tmp/vestigium

downloaddir = /usr/vestigium/downloads

wrapperpath = /usr/vestigium/wrappers

#The packagetype flag allows you to set what
#type of packages you wish to use. You can use
#your backups (if you created them). You can use
#binary packages, and you can build packages
#from source.
packagetype = source
#pacakgetype = binary
#packagetype = backup

#Set your architechture below. This will be used to
#automatically determine the correct binary packages
#to download and determine which ports to download.
#Currently, the only availible architechture is x86.
arch = x86

#Set your subarchitechture (e.g. Pentium4 or AthlonXP)
#below. This will be used to bring in optimized binaries
#or optimize source during build time. Currently
#availible subarchs are athlon64 and i686.
subarch = athlon64-32

#If you are not happy with the default flags that
#will automatically be set for you with the arch and
#subarch flags, you can write your own flags if you
#know what you are doing. Note however, that you will
#continue to download binary packages according to your
#arch and subarch. Therefore this is practical only for
#source packages. You have encapsulate the following in
#double quotes. The will over-ride what ever is set my
#arch and subarch.
#chost =
#cflags =
#cxxflags =
#ldflags =

#Set "mirror" to the url of the mirror from where
#you wish to download your packages and ports from.
#You set a list of mirrors by separating them with
#spaces. The first one in the list will take
#precedence. All of them have to be on the same
#line.
mirror = http://afalko.homelinux.net/vestigium

#If you wish to prevent Vestigium from fetching
#the latest ports and other such stuff you can
#set nosync to true. Default is false.
nosync = false

#If you wish to be informed of when Vestigium
#enters and exits subroutines, set debug to true.
debug = true

#If prompt is set to true Vestigium will show what
#it plans to do and allow the user to say yes to no.
#Setting prompt to false will prevent Vestrigium
#from doing the above. true is the default.
prompt = false

#When removing a package, it might have dependencies,
#which will not be needed by any other package. Setting
#cleandeps to false will prevent this "cleaning" process
#from occurring. The default is true.
cleandeps = true

#Vestigium has a special file collision handling system.
#By default, when a program is being installed and is
#attempting to write a file that already exists, it
#searchs the for which package the file on the system
```

```
#belongs. Once if finds that package, it will back the
#file up under that package's backup directory (by default
#/var/spool/vestigium/<package name>). The incoming file
#will get written once the existing file is backed up.
#The backed up file is called duplicate, or dupe for short.
#We set two things here with dupes:
#
#1) alwaysforce. This prevent the dupes mechanism from
#functioning. When a file collision occurs, existing files
#become overwritten. Default for this is false.
alwaysforce = false

#2) flipdupes. Set this to true prevent the incoming file
#from being written, and instead make it the file that
#Vestigium is going to back up. The default is false.
flipdupes = false

#By default, Vestigium's --search option will search
#package names only. The following flag changes this behavior
#to also search dependencies whenever --search flag is passed.
#The default is false.
alwayssearchdeps = false

#Createbinaries make Vestigium create binary package upon a
#package's installation (also installing the package). Default
#is false.
createbinaries = false

#If you wish to create binary packages without installing package,
#set onlycreatebinaries to true. The default is set to false.
onlycreatebinaries = false
```

C.1.11 subarchs

```
athlon64
        chost = "x86_64-pc-linux-gnu"
        cflags = "-O3 -march=athlon64 -ftracer -pipe"
        cxxflags = "-O3 -march=athlon64 -ftracer -pipe"
end

athlon64-32
        chost = "i686-pc-linux-gnu"
        cflags = "-O3 -march=athlon64 -ftracer -pipe"
        cxxflags = "-O3 -march=athlon64 -ftracer -pipe -fvisibility-inlines-hidden"
end
```

C.1.12 vesport

```
#!/usr/bin/perl -w

$configFile = "/etc/vestigium.conf";

open CONFIG, $configFile
        or die "Unable to open the configuration file: $!";
#First filter out the #Comment lines.
while (<CONFIG>) {
        chomp $_;
        #print "$_\n" if (! /^#/ && !($_ eq ""));
        push @options, $_ if (! /^#/ && !($_ eq ""));
}
close CONFIG;
#Set the varibles.
SWITCH: for (@options) {
                #/^libdir\s*=\s*(\S+)/           && do { $libDir = $1; next };
                /^portdir\s*=\s*(\S+)/          && do { $portDir = $1; next };
                #do { warn "Warning: I did not understand $_ in $configFile." };
        }

use lib "/usr/lib/vestigium";
require "names.lib";

print "Vesport is a utility that will help you create ports for the Vestigium package manager.
answer the following question and all necessary files will be created for your port.\n";

while (1) {
        print "Enter the program name (no version attached): ";
        chomp ($prog = <STDIN>);
        if ($prog =~ /\-\d/) {
                print "Program names cannot have digits after dashes. Sorry, try again.\n";
                next;
        } elsif ($prog =~ /\s/) {
                print "I detected white space. This is no good, try again.\n";
                next;
        } else {
                last;
        }
}

while (1) {
        print "Enter the program version (without a starting dash): ";
        chomp (my $ver = <STDIN>);
        if ($ver =~ /^\-/) {
                print "Follow instructions. No leading dashes! Try again.\n";
                next;
        } elsif (! $ver =~ /^\d/) {
                print "I regret to inform you that program versions can only start with digits. Try again.\n";
                next;
        } else {
                $progVer = "$prog-$ver";
                last;
        }
}

while (1) {
        print "Enter a description for the program. Be brief please (pressing enter will end the description): ";
        chomp ($desc = <STDIN>);
        if (! $desc) {
                print "You entered nothing, I am going to ask you to try again.\n";
                next;
        } else {
                last;
        }
}

while (1) {
        print "Enter the licence under which this program licensed: ";
        chomp ($license = <STDIN>);
        if (! $license) {
                print "You entered nothing, I am going to ask you to try again.\n";
                next;
        } else {
                last;
        }
}


while (1) {
        print "Enter the homepage of this program: ";
        chomp ($homepage = <STDIN>);
        if (! $homepage) {
                print "You entered nothing, I am going to ask you to try again.\n";
                next;
        } elsif ($homepage =~ /none/) {
                last;
        } elsif (! $homepage =~ /^(http|ftp)/) {
                print "Enter an url starting with either ftp or http.\n";
                next;
        } else {
```

```
                last;
        }
}

while (1) {
        print "Please enter where the source code files are to be downloaded from (one per line; type \"done\" when done): ";
        while (<STDIN>) {
                chomp;
                last if (/done/);
                if (! /^(http|ftp)/) {
                        print "Enter an url starting with either ftp or http.\n";
                        next;
                } else {
                        push @sources, $_;
                        next;
                }
        }
        last;
}

while (1) {
        print "Enter the required dependencies of the program (one per line; type \":q\" when done): ";
        while (<STDIN>) {
                chomp;
                last if (/:q/);
                push @deps, $_;
                next;
        }
        last;
}

while (1) {
        print "Enter the program's default CFLAGS (this will over-ride the global CFLAGS;
CXXFLAGS will be set to the same thing; entering this is optional): ";
        chomp ($cflags = <STDIN>);
        my @flags = split " ", $cflags;
        for (@flags) {
                if (/-[a-zA-Z0-9]+/) {
                        last;
                } elsif (! $_) {
                        last;
                } else {
                        print "As far as I know, All CFLAG flags are supposed to start with a single \"-\". Try again.\n";
                        next;
                }
        }
        last;
}

while (1) {
        print "Enter the program's option flags (one per line; type \":q\" when done): ";
        while (<STDIN>) {
                chomp;
                last if (/:q/);
                push @opts, $_;
                next;
        }

        for (@opts) {
                print "Enter the dependencies which will be brought in for $_ (one per line; type \":q\" when done): ";
                while (<STDIN>) {
                        last if (/:q/);
                        push @temp, $_;
                        next;
                }
                $opts{$_} = \@temp;
                @temp = undef;

                print "Enter the configure options for $_ (one per line; type \":q\" when done): ";
                while (<STDIN>) {
                        last if (/:q/);
                        push @temp, $_;
                        next;
                }
                $opts{$_}{$_} = \@temp;
                @temp = undef;
        }
        last;
}

while (1) {
        print "Enter the commands to execute from extraction to right before installation (one per line; type \":q\" when done): ";
        while (<STDIN>) {
                chomp;
                last if (/:q/);
                push @build, $_;
                next;
        }
        last;
}

while (1) {
```

```
        print "Enter the commands to execute after building, but before post-installation (one per line; type \":q\" when done): ";
        while (<STDIN>) {
                chomp;
                last if (/:q/);
                push @install, $_;
                next;
        }
        last;
}

while (1) {
        print "Enter the commands to execute after installation (one per line; type \":q\" when done; this is optional): ";
        while (<STDIN>) {
                chomp;
                last if (/:q/);
                push @postinst, $_;
                next;
        }
        last;
}

print "It appears as if you gave me enough information.\nHere is what I plan to do: \n";

$let = &getFirstLet($prog);
print "Create port in $portDir/$let/$prog/$progVer\n";

print "Create file \"$portDir/$let/$prog/$progVer/BUILD\" containing the following lines: \n";
for (@build) {
        print "     $_\n";
}

print "Create file \"$portDir/$let/$prog/$progVer/INSTALL\" containing the following lines: \n";
for (@install) {
        print "     $_\n";
}

print "Create file \"$portDir/$let/$prog/$progVer/POSTINST\" containing the following lines: \n";
for (@postinst) {
        print "     $_\n";
}

print "Create file \"$portDir/$let/$prog/$progVer/INFO\" containing the following lines: \n";
print "     description = \"$desc\"\n";
print "     licence = \"$license\"\n";
print "     homepage = \"$homepage\"\n";
$sources = join "          \n", @sources;
print "     files = \"\n$sources\"\n";
$deps = join "         \n", @deps;
print "     deps = \"\n$deps\"\n";
print "     options = {\n";
for (sort keys %opts) {
        my $deps = join " ", @$opts{$_} if (! $opts{$_}{$_});
        print " " x 12;
        if (! $opts{$_}{$_}) {
                print "$_; @$opts{$_}; ";
                $conf = join " ", $opts{$_}{$_};
                print "$conf\n";
        }
}

print "Would you like me to continue with all of this? (N/y): ";
chomp ($answ = <STDIN>);
if ($answ =~ m/y/i) {
        #last;
} else {
        exit 0;
}

system "mkdir -p $portDir/$let/$prog/$progVer";
open BUILD, ">>$portDir/$let/$prog/$progVer/BUILD" or die "Could not open file handle: $!";
select BUILD;
for (@build) {
        print "$_\n";
}

open INST, ">>$portDir/$let/$prog/$progVer/INSTALL";
select INST;
for (@install) {
        print "$_\n";
}

open POSTINST, ">>$portDir/$let/$prog/$progVer/POSTINST";
select POSTINST;
for (@postinst) {
        print "$_\n";
}

open INFO, ">>$portDir/$let/$prog/$progVer/INFO";
select INFO;
print "description = \"$desc\"\n";
print "licence = \"$license\"\n";
print "homepage = \"$homepage\"\n";
```

```
$sources = join "\n", @sources;
print "files = \"$sources\"\n";
$deps = join "\n", @deps;
print "deps = \"$deps\"\n";
#print "opts = \"
```

```
install:
	groupadd vestigium
	cp vestigium /usr/bin/vestigium
	cp vesport /usr/bin/vesport
	chown root:vestigium /usr/bin/vesport
	chmod 775 /usr/bin/vesport
	chown root:vestigium /usr/bin/vestigium
	chmod 775 /usr/bin/vestigium
	mkdir -p /usr/lib/vestigium
	chown root:vestigium /usr/lib/vestigium
	chmod 775 /usr/lib/vestigium
	cp *.lib /usr/lib/vestigium/
	cp subarchs /usr/lib/vestigium/
	mkdir -p /usr/vestigium/wrappers
	cp wrappers/* /usr/vestigium/wrappers/ -R
	chown root:vestigium /usr/vestigium -R
	chmod 775 /usr/vestigium -R
	cp vestigium.conf /etc/vestigium.conf
	chown root:vestigium /etc/vestigium.conf
	chmod 775 /etc/vestigium.conf
	install -o root -g vestigium -m 775 -d /tmp/vestigium
	install -o root -g vestigium -m 775 -d /var/log/vestigium
	install -o root -g vestigium -m 775 -d /var/spool/vestigium
	ln -s /usr/bin/vestigium /usr/bin/ves

update:
	cp vestigium /usr/bin/vestigium
	chown root:vestigium /usr/bin/vestigium
	chmod 775 /usr/bin/vestigium
	cp vesport /usr/bin/vesport
	chown root:vestigium /usr/bin/vesport
	chmod 775 /usr/bin/vesport
	chown root:vestigium /usr/lib/vestigium
	chmod 775 /usr/lib/vestigium
	cp *.lib /usr/lib/vestigium/
	cp subarchs /usr/lib/vestigium/
	cp wrappers/* /usr/vestigium/wrappers/ -R
	cp ports/* /usr/vestigium/ports/ -R
	chown root:vestigium /usr/vestigium -R
	chmod 775 /usr/vestigium -R
	cp vestigium.conf /etc/vestigium.conf
	chown root:vestigium /etc/vestigium.conf
	chmod 775 /etc/vestigium.conf

package:
	mkdir temp
	cp /usr/bin/vestigium temp/
	cp /usr/bin/vesport temp/
	cp /usr/vestigium/* temp/ -R
	rm -Rf temp/ports
	rm -Rf temp/downloads
	cp /etc/vestigium.conf temp/
	cp /usr/lib/vestigium/* temp/
	cp Makefile temp/
	tar -cjf vestigium.tar.bz2 temp

unpack:
	tar -xf ~/vestigium.tar.bz2 -C /tmp/
	cp /tmp/temp/* ../vestigium/ -R
	rm *~
	rm /tmp/temp -R
	rm ~/vestigium.tar.bz2

clean:
	rm temp -R
	rm vestigium.tar.bz2

remove:
	groupdel vestigium
	rm /usr/bin/vestigium
	rm /usr/bin/vesport
	rm -R /usr/vestigium
	rm /etc/vestigium.conf
	rm -R /tmp/vestgium
	rm -R /var/log/vestigium
	rm -R /var/spool/vestigium
```

C.2 Sysmark

C.2.1 sysmark

```
#!/bin/bash
#sysmark - a collection of benchmarking tools for GNU/Linux.
#Copyright (C) 2006  Andrey Falko
#
#This program is free software; you can redistribute it and/or
#modify it under the terms of the GNU General Public License
#as published by the Free Software Foundation; either version 2
#of the License, or (at your option) any later version.
#
#This program is distributed in the hope that it will be useful,
#but WITHOUT ANY WARRANTY; without even the implied warranty of
#MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#GNU General Public License for more details.
#
#You should have received a copy of the GNU General Public License
#along with this program; if not, write to the Free Software
#Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA.
#
#       You may contact me, Andrey Falko, at Ma3oxuct@gmail.com
#

SRCDIR=$PWD #The directory that sysmark is in.
TMPDIR=/usr/tmp
OUTFILE=$SRCDIR/results.txt

if [ -a $OUTFILE ]
        then
        echo "You have ran this benchmark before and have results.
        Would you like me to overwrite them?"
        read in
        if [ "$in" = 'n' ]
                then
                echo "Exiting."
                exit 0
        else
                rm $OUTFILE
        fi
fi

help="The following command line options are availible: \n
--no-scimark \t\t      Excludes the Scimark benchmark \n
--no-bashmark \t\t    Excludes the Bashmark benchmark \n
--no-lame \t\t        Excludes the Lame benchmark \n
--no-pov \t\t         Excludes the Pov-Ray benchmark \n
--no-interbench \t  Excludes the Interbench benchmark \n
--no-lmbench \t\t     Excludes the lmbench benchmark \n
\n
--only-[benchmark name]  Includes only [benchmark name] \n
\n
--live (-l) \t\t      Provides live output of the benchmarks in progress \n
--full (-f) \t\t      Provides output from the benchmarks all together at the end \n
--summary (-s) \t   Provides shorthanded output at the end (default)\n
\n
All results are stored in $OUTFILE"

SCIMARK=true
BASHMARK=true
LAME=true
POV=true
INTER=true
LMBENCH=true

LIVE=false
FULL=false
SUMMERIZE=true
while [ $1 ]; do
        case  $1 in
        "-h" | "--help")
                echo -e $help
                exit 0;;
        "--no-scimark")
                SCIMARK="false";;
        "--no-bashmark")
                BASHMARK="false";;
        "--no-lame")
                LAME="false";;
        "--no-pov")
                POV="false";;
        "--no-interbench")
                INTER="false";;
        "--no-lmbench")
                LMBENCH="false";;
        "--only-scimark")
                BASHMARK="false"
                LAME="false"
                POV="false"
                INTER="false"
```

```
                LMBENCH="false";;
        "--only-bashmark")
                SCIMARK="false"
                LAME="false"
                POV="false"
                INTER="false"
                LMBENCH="false";;
        "--only-lame")
                SCIMARK="false"
                BASHMARK="false"
                POV="false"
                INTER="false"
                LMBENCH="false";;
        "--only-pov")
                SCIMARK="false"
                BASHMARK="false"
                LAME="false"
                INTER="false"
                LMBENCH="false";;
        "--only-interbench")
                SCIMARK="false"
                BASHMARK="false"
                LAME="false"
                POV="false"
                LMBENCH="false";;
        "--only-lmbench")
                SCIMARK="false"
                BASHMARK="false"
                LAME="false"
                POV="false"
                INTER="false";;
        "-l" | "--live")
                LIVE="true"
                SUMMERIZE="false";;
        "-f" | "--full")
                FULL="true"
                SUMMERIZE="false";;
        "-s" | "--summary")
                SUMMERIZE="true";;
        "");;
        *)
                echo -e $help
                exit 0;;
        esac
        shift
done

cd $SRCDIR

SYS=`echo | ./config.guess`

touch $OUTFILE
echo "Welcome to the sysmark benchmarking suite"

if [ -z "${CFLAGS}" ] && [ -z "${CXXFLAGS}" ]
        then
        echo "You have not yet set your optimization flags. Set them now: "
        read flags

        if [ -z "${flags}" ]
                then
                echo "You entered nothing, assuming -O0."
                export CFLAGS="-O0"
        else
                export CFLAGS=$flags
        fi
fi
if [ -n "${CFLAGS}" ]
        then
        export CXXFLAGS=$CFLAGS
elif [ -n "${CXXFLAGS}" ]
        then
        export CFLAGS=$CXXFLAGS
else
        echo "Error: All flags are null. $CFLAGS - $CXXFLAGS"
fi

echo "The CFLAGS are: $CFLAGS"

if [ $LAME = "true" ]
        then
        echo "Sysmark will require you to have to 500 megabytes of temporary free space."
        echo "Please input your desired temporary directory. Make sure that it has write access (default is $TMPDIR): "
        read dir
        if [ -n "${dir}" ]
                then
                TMPDIR=$dir
        fi
fi

if [ $SCIMARK = "true" ]
        then
```

```
        echo "Comiling scimark2..."
        cd $SRCDIR/scimark
        make scimark2 > /dev/null 2>&1
        echo "Finished compiling, running benchmark..."
        if [ $LIVE = "true" ]
                then
                ./scimark2
        else
                ./scimark2 >> $OUTFILE
        fi
        make clean > /dev/null 2>&1
        echo "Scimark finished."
fi

if [ $BASHMARK = "true" ]
        then
        echo "Compiling bashmark..."
        cd $SRCDIR/bashmark
        make bashmark > /dev/null 2>&1
        echo "Finished compiling bashmark, running benchmark..."
        if [ $LIVE = "true" ]
                then
                ./bashmark
        else
                ./bashmark >> $OUTFILE
        fi
        make clean > /dev/null 2>&1
        echo "Bashmark finished."
fi

if [ $LAME = "true" ]
        then
        echo "Comiling Lame..."
        cd $SRCDIR/lame
        ./configure > /dev/null 2>&1
        make > /dev/null 2>&1
        echo "Finished compiling Lame, generating 500 megabyte wav file..."
        i=1
        while [ $i -le 205 ]
        do
                cp wavfile $TMPDIR/wavfile.$i
                i=$((i+1))
        done
        cat $TMPDIR/wavfile.* > $TMPDIR/bigwav.wav
        echo "File generated. Encoding it with Lame..."
        cd frontend
        if [ $LIVE = "true" ]
                then
                ./lame -hr $TMPDIR/bigwav.wav /dev/null #-r so that lame does not cheat.
        else
                ./lame -hr $TMPDIR/bigwav.wav /dev/null >> $OUTFILE 2>&1
        fi
        cd ../
        make clean > /dev/null 2>&1
        rm $TMPDIR/wavfile.* $TMPDIR/bigwav.wav
        echo "Lame finished."
fi

if [ $POV == "true" ]
        then
        echo "Compiling Pov-Ray..."
        cd $SRCDIR/povray
        ./configure --prefix=$SRCDIR/povray/inst COMPILED_BY="sysmark" > /dev/null 2>&1
        make > /dev/null 2>&1
        make install > /dev/null 2>&1
        cd inst/share/povray-3.6/scenes/advanced
        echo "Finished compiling Pov-Ray, running benchmark..."
        if [ $LIVE = "true" ]
                then
                $SRCDIR/povray/inst/bin/povray benchmark.ini
        else
                $SRCDIR/povray/inst/bin/povray benchmark.ini >> $OUTFILE 2>&1
        fi
        cd $SRCDIR/povray
        rm inst/* -R
        make clean > /dev/null 2>&1
        echo "Pov-Ray finished."
fi

if [ $INTER == "true" ]
        then
        cd $SRCDIR/interbench
        echo "Comiling interbench..."
        make interbench > /dev/null 2>&1
        echo "Finished compiling Interbench, running benchmark..."
        if [ $LIVE = "true" ]
                then
                ./interbench
        else
                ./interbench >> $OUTFILE
        fi
        make clean > /dev/null 2>&1
```

```
        echo "Interbench finished."
fi

if [ $LMBENCH == "true" ]
        then
        cd $SRCDIR/lmbench/src
        echo "Compiling lmbench..."
        #We do not want this benchmark to use any optimization flags...at least for now.
        make lmbench > /dev/null 2>&1
        echo "lmbench compiled, running benchmark..."
        if [ -a $SRCDIR/lmbench/CONFIG.localhost ]
                then
                echo "A mandatory configuration file has been located. Would you like to
generate a new one? (Say yes, unless you know what you are doing)."
                read in
                if [ $in = 'n' ]
                        then
                        cp $SRCDIR/lmbench/CONFIG.localhost $SRCDIR/lmbench/bin/$SYS/CONFIG.$HOSTNAME
                        make os > /dev/null 2>&1
                else
                        make os > /dev/null 2>&1
                fi
        else
                make os > /dev/null 2>&1
                cp $SRCDIR/lmbench/bin/$SYS/CONFIG.$HOSTNAME $SRCDIR/lmbench/CONFIG.localhost
        fi
        i=-1
        for t in $SRCDIR/lmbench/results/$SYS/*
        do
                i=$((i+1))
        done
        if [ $LIVE = "true" ]
                then
                cat $SRCDIR/lmbench/results/$SYS/$HOSTNAME.$i
        else
                cat $SRCDIR/lmbench/results/$SYS/$HOSTNAME.$i >> $OUTFILE
        fi
        make clean > /dev/null 2>&1
        echo "lmbench finished."
fi

echo "All benchmarks have finished."

if [ $FULL = "true" ]
        then
        export OUTFILE=$OUTFILE
        cat $OUTFILE
else
        $SRCDIR/parse
fi
```

C.2.2 parse

parse

```
#!/bin/bash
OUTFILE=$PWD/results.txt
echo "Parsing results..."
if [ -a $OUTFILE ]
        then
        cat $OUTFILE
else
        echo "Error: File not found. Run sysmark first?"
fi
```

C.2.3 README

```
This is sysmark, a collection of benchmarking tools for GNU/Linux. It was written by Andrey Falko for his
thesis project in Linux Distributions. The License underwhich this programs is under, is GPL v2.

Installation:
        The simplest thing to do is
                tar -xjf sysmark-1.0.1.tar.bz2
                cd sysmark-1.0.1

        Extract the source.
        You can also do
                make install
        This will install the program to /usr/src/sysmark. You will be able to execute the script from anywhere,
        but will have to do so only as root (unless you edit the Makefile for your preferences).

Usage:
        You should be able to safely execute sysmark with
                cd <dir that you extracted it to>
                ./sysmark

        You can do
                ./sysmark -h or ./sysmark --help to see additional options that you can apply.

Changelog:
        May 21, 2006 (Current)
                Version 1.0.2
                -Added feature to determine the target triplet.
                -Added the --only commandline options.
                -Fixed bug

        May 20, 2006
                Version 1.0.1
                -Fixed a few bugs and typos

        May 19, 2006
                Version 1.0.0
                -A few bug fixes.
                -lmbench is in the suite.
                -sysmark is ready for primetime.

        May 19, 2006
                Version 0.2.0
                -Lots of bug fixes and improvments.
                -Added help command line option.
                -Added an easy way to chose which benchmarks not to run.
                -Added output options.

        May 18, 2006
                Version 0.1.0 of sysmark, the first version.

Current todo list:
        Make fancy results parsing.
```

C.2.4 Makefile

```
SRCDIR=${PWD}

run:
        echo "You can just run ./sysmark."
        ./sysmark

install:
        mkdir /usr/src/sysmark
        cp * /usr/src/sysmark/ -R
        cp sysmark /usr/sbin/sysmark

uninstall:
        rm /usr/src/sysmark -R
        rm /usr/sbin/sysmark

clean:
        #cd ${SRCDIR}/scimark
        #make clean
        #cd ${SRCDIR}/bashmark
        #make clean
        #cd ${SRCDIR}/lame
        #make clean
        #cd ${SRCDIR}/povray
        #rm inst/* -R
        #make clean
        #cd ${SRCDIR}/interbench
        #make clean
        #cd ${SRCDIR}/lmbench/src
        #make clean
        rm ${SRCDIR}/results.txt
        rm -f ${SRCDIR}/lmbench/CONFIG.localhost
        rm -rf ${SRCDIR}/lmbench/results/*
```

C.3 Simulation

C.3.1 main

```
#!/usr/bin/perl
#
# Andrey Falko
# Simulation
use strict;
use vars qw [ $inf $T $weeks %pkgCompiles %pkgMaintainers ];

# These are the results that we got from out data:
my $avgVerBump = 3109.32494279176;
my $avgBugRep = 11829.4724547401;
# Alpha determines how many recompiles before commit
# of a package; for out gamma generator.
my $alpha = 1;
my $beta = 1;

# We assume that it take a maintainer 1.5 hours to
# prepare a package before proceeding to compile it
my $prep = &genPrepTime;

# Set some constants
$inf = 1000000000;
$T =   31536000;
#$T = 200000;

# Load data corresponding to packages and thier compile times.
open PKGCOMP, 'pkg-comp-times';
for (<PKGCOMP>) {
        /^(.+): (\d+)/;
        $pkgCompiles{$1} = $2;
}
close PKGCOMP;

# Load data regarding which packages belong to maintainer.
open PKGMAINT, 'maintainers' or die "Could not open: $!";
for (<PKGMAINT>) {
        /^(.+): (.+)/;
        $pkgMaintainers{$1} = $2;
}
close PKGMAINT;

#my $t = 0;
#my @compiles;
#while (1) {
#       my $vB = &genVersionBump($t);
#       print "VerBump: $vB\n";
#       my $bB = &genBugReport($t);
#       print "BugBump: $bB\n";
#       my $compiles = 1 + &genNumberOfCompiles;
#       print "Compiles: $compiles\n";
#       push @compiles, $compiles;
#       my $response = &genResponseTime($t);
#       print "Response Time: $response\n";
#       $t = &min($vB, $bB);
#       if ($t > $T) {
#               last;
#       }
#}

#my $temp = &sum(\@compiles);

#print " ". $temp/@compiles ."\n";
#exit 0;

open LOG, "> main.log" or die "$!";
select LOG;

# Number of Simulation runs
my $runs = 10;

$weeks = ($T / 604800);

# Statistical lists
my @meanWWT;
my @meanDH;
my @meanUN;
my @bumps;
my @bugs;

for (1..$runs) {
        my $t = 0;
        my $n = 0; # Number of packages presently being worked on

        # Occurrances of respective events
        my $tAVer = &genVersionBump($t);
        my $tABug = &genBugBump($t);

        # Packages that we can choose from
```

```
my @pkgAvail = keys %pkgCompiles;
# Packages that we cannot choose from and the time
# when they will be added to @pkgAvail
my %releasePkg;
# Next time of package release
my $tR = $inf;

# Packages' commital chart
my %commitPkg;
# Time when next package will finish service
my $tC = $inf;

# Next time of package release
my $tR = $inf;

# Statistical counters
my %pkgTotalServiceTime;
my %pkgTimesServed;
my $numberOfBumps;
my $numberOfBugs;
my $totalCommits;
my %pkgWeeklyServiceTime;

while (1) {
        print "Time is: $t\n";

        # Find $tR
        for (sort {$a <=> $b } keys %releasePkg) {
                $tR = $_;
                last;
        }

        for (sort {$a <=> $b } keys %commitPkg) {
                $tC = $_;
                last;
        }

        # Case 1: Version Bump occurs
        if ($tAVer == &min($tAVer, $tABug, $tC, $tR)) {
                $t = $tAVer;
                $n = $n + 1;
                print "Version Bump starts at time $t";
                $tAVer = &genVersionBump($t);
                if ($tABug < $tAVer) {
                        $tABug = &genBugBump($t);
                }
                my $choice = int(rand() * @pkgAvail);
                my $pkg = $pkgAvail[$choice];
                print " for $pkg\n";

                # Remove $pkg from pkgAvail
                my @temp;
                for (@pkgAvail) {
                        push @temp, $_ unless ($_ eq $pkg);
                }
                @pkgAvail = @temp;

                my $compiles = &genNumberOfCompiles;
                my $prep = &genPrepTime;
                my $c = $compiles*($prep + $pkgCompiles{$pkg});
                print "$pkg to be committed at $c\n";
                my $temp = $c + $t;
                $commitPkg{$temp} = $pkg;

                my $r = &genReleaseTime($t);
                print "$pkg will be released at time $r\n";
                $releasePkg{$r} = $pkg;

                # Collect statistics
                $pkgTotalServiceTime{$pkg} = $pkgTotalServiceTime{$pkg} + $c;
                $pkgTimesServed{$pkg} = $pkgTimesServed{$pkg} + 1;
                $numberOfBumps = $numberOfBumps + 1;
        }

        # Case 2: Bug occurs
        if ($tABug == &min($tAVer, $tABug, $tC, $tR)) {
                $t = $tABug;
                $n = $n + 1;
                print "Bug Bump starts at time $t";
                $tABug = &genBugBump($t);
                if ($tAVer < $tABug) {
                        $tAVer = &genVersionBump($t);
                }
                my $choice = int(rand() * @pkgAvail);
                my $pkg = $pkgAvail[$choice];
                print " for $pkg\n";

                # Remove $pkg from pkgAvail
                my @temp;
                for (@pkgAvail) {
                        push @temp, $_ unless ($_ eq $pkg);
                }
```

```
                @pkgAvail = @temp;

                my $compiles = &genNumberOfCompiles;
                my $prep = &genPrepTime;
                my $c = $compiles*($prep + $pkgCompiles{$pkg});
                print "$pkg to be committed at $c\n";
                my $temp = $c + $t;
                $commitPkg{$temp} = $pkg;

                my $r = &genReleaseTime($t);
                print "$pkg will be released at time $r\n";
                $releasePkg{$r} = $pkg;

                # Collect statistics
                $pkgTotalServiceTime{$pkg} = $pkgTotalServiceTime{$pkg} + $c;
                $pkgTimesServed{$pkg} = $pkgTimesServed{$pkg} + 1;
                $numberOfBugs = $numberOfBugs + 1;
        }

        # Case #3: Package Committed
        if ($tC == &min($tAVer, $tABug, $tC, $tR)) {
                $t = $tC;
                my $pkg = $commitPkg{$t};
                $n = $n - 1;
                print "$pkg Committed at time: $t\n";
                if (! (keys %commitPkg) && $n != 0) {
                        print 'n should by 0, but it not'."\n";
                }
                delete $commitPkg{$t};

                for (keys %commitPkg) {
                        print "Debug: $commitPkg{$_}\n" if ($n == 1);
                }
                $tC = $inf;

                $totalCommits = $totalCommits + 1;
        }

        # Case #4: Package gets released
        if ($tR == &min($tAVer, $tABug, $tC, $tR) && (keys %releasePkg) > 0) {
                $t = $tR;
                my $pkg = $releasePkg{$t};
                print "Releasing $pkg at time $t\n";
                delete $releasePkg{$t};
                $tR = $inf;
                for (@pkgAvail) {
                        if ($_ eq $pkg) {
                                die 'Bug: Package already in @pkgAvail';
                        }
                }
                push @pkgAvail, $pkg;;
        }

        print "Number of packages in service: $n\n";

        if (&min($tAVer, $tABug, $tC) > $T && $n == 0) {
                last;
        }

}

printf STDOUT "Number of version bumps: $numberOfBumps\n";
printf STDOUT "Number of bug bumps: $numberOfBugs\n";
printf STDOUT "Total number of commits: $totalCommits\n";

push @bumps, $numberOfBumps;
push @bugs, $numberOfBugs;

for (keys %pkgTotalServiceTime) {
        #printf STDOUT "Total service time for $_: $pkgTotalServiceTime{$_} in $pkgTimesServed{$_} times served\n";
        $pkgWeeklyServiceTime{$_} = $pkgTotalServiceTime{$_} / $weeks;
        #printf STDOUT "Weekly service time for $_: $pkgWeeklyServiceTime{$_}\n";
}

# Statistical Compuations

my $total = 0;
for my $pkg (keys %pkgWeeklyServiceTime) {
        $total = $total + $pkgWeeklyServiceTime{$pkg};
}

my $meanWeeklyWorkTime = $total / $weeks;
printf STDOUT "Mean work done per week: $meanWeeklyWorkTime\n";
push @meanWWT, $meanWeeklyWorkTime;

my $dhTotalWeeklyWorkTime;
my $unTotalWeeklyWorkTime;
my @dhWWT;
my @unWWT;

for my $pkg (keys %pkgMaintainers) {
        if ($pkgMaintainers{$pkg} eq 'dragonheart') {
```

```
                        $dhTotalWeeklyWorkTime = $dhTotalWeeklyWorkTime + $pkgWeeklyServiceTime{$pkg};
                        push @dhWWT, $pkgWeeklyServiceTime{$pkg};
                        print "Daniel \"Dragonheart\" Black maintains ". @dhWWT ."packages\n"
                } elsif ($pkgMaintainers{$pkg} eq 'unknown') {
                        $unTotalWeeklyWorkTime = $unTotalWeeklyWorkTime + $pkgWeeklyServiceTime{$pkg};
                        push @unWWT, $pkgWeeklyServiceTime{$pkg};
                        print "Everyone else maintains ". @unWWT ."packages\n"
                } else {
                        die "Bug: $pkg not allocated up!";
                }
        }

        my $dragonheartavg = $dhTotalWeeklyWorkTime / $weeks;
        printf STDOUT "Daniel \"Dragonheart\" Black works $dragonheartavg seconds per week\n";
        push @meanDH, $dragonheartavg;

        my $unknownavg = $unTotalWeeklyWorkTime / $weeks;
        printf STDOUT "Everyone else works $unknownavg seconds per week\n";
        push @meanUN, $unknownavg;
}

select STDOUT;

my $total = 0;
for (@bumps) {
        $total = $total + $_;
}

my $avgBumps = $total / @bumps;
print "Average number of version bumps: $avgBumps\n";

$total = 0;
for (@bumps) {
        $total = $total + ($_ - $avgBumps)**2;
}

my $varBumps = $total / (@bumps - 1);
print "Variance of version bumps: $varBumps\n";

my $total = 0;
for (@bugs) {
        $total = $total + $_;
}

my $avgBugs = $total / @bugs;
print "Average number of version bumps: $avgBugs\n";

$total = 0;
for (@bugs) {
        $total = $total + ($_ - $avgBugs)**2;
}

my $varBugs = $total / (@bugs - 1);
print "Variance of version bumps: $varBugs\n";

$total = 0;
for (@meanWWT) {
        $total = $total + $_;
}

my $meanWeeklyWorkTime = $total / @meanWWT;
print "Aggregate mean weekly work time: $meanWeeklyWorkTime\n";

$total = 0;
for (@meanWWT) {
        $total = $total + ($_ - $meanWeeklyWorkTime)**2;
}

my $varWeeklyWorkTime = $total / (@meanWWT - 1);
print "Variance of aggregate work done per week: $varWeeklyWorkTime\n";

$total = 0;
for (@meanDH) {
        $total = $total + $_;
}

my $meanDH = $total / @meanDH;
print "Mean weekly work time per year for Dragonheart: $meanDH\n";

$total = 0;
for (@meanDH) {
        $total = $total + ($_ - $meanDH)**2;
}

my $varDH = $total / (@meanDH - 1);
print "Variance: $varDH\n";

$total = 0;
for (@meanUN) {
        $total = $total + $_;
}
```

```
my $meanUN = $total / @meanUN;
print "Mean weekly work time per year for everyone else: $meanUN\n";

my $total = 0;
for (@meanUN) {
        $total = $total + ($_ - $meanUN)**2;
}

my $varUN = $total / (@meanUN - 1);
print "Variance: $varUN\n";

sub genVersionBump {
        my $t = shift;
        while (1) {
                my $U = rand;
                $t = $ase it occurs more
        # frequently
        my $t = shift;
        return &genVersionBump($t);
}

sub genNumberOfCompiles {
        # This is the gamma distribution with alpha = 1, beta = 1.
        # After we get the gamma value, we shift it by one.
        my $d = $alpha - (1/3);
        my $c = 1 / sqrt(9*$d);
        while (1) {
                my $X = &genStandardNormal;
                my $v;
                $v = (1 + $c * $X)**-3;
                redo if ($v <= 0);

                my $U = rand;
                if ($U < $beta - .0331*$X**4) {
                        my $dist = $d * $v;
                        return $dist;
                }
                if (log($U) < .5 * $X**2 + $d*(1 - $v + log($v))) {
                        my $dist = $d * $v;
                        return $dist + 1;
                }
        }
}

sub genStandardNormal {
        # This algorithm is taken from the Perl Cookbook; Chapter 2.
        # This outputs standard normals with mean 0, standard deviation 1.

        my ($u1, $u2);  # uniformly distributed random numbers
        my $w;          # variance, then a weight
        my ($g1, $g2);  # gaussian-distributed numbers

        do {
                $u1 = 2 * rand() - 1;
                $u2 = 2 * rand() - 1;
                $w = $u1*$u1 + $u2*$u2;
        } while ( $w >= 1 );

        $w = sqrt( (-2 * log($w))  / $w );
        $g2 = $u1 * $w;
        $g1 = $u2 * $w;
        # return both if wanted, else just one
        return wantarray ? ($g1, $g2) : $g1;
}

sub genPrepTime {
        my $time = rand;
        my $hours = 5 * rand();
        $time = 5*60 + ($time * $hours*60*60);
        return $time;
}

sub min {
        my @array = sort { $b <=> $a } @_; # Sort the input array numerically, but descending.
        return pop @array; # Return the smallest value, which will be on top of the array.
}

sub sum {
        my $array = shift;
        my $total = 0;
        for (@$array) {
                $total = $total + $_;
        }
        return $total;
}
```

C.3.2 maintainers-create

```
#!/usr/bin/perl

open DRAGONHEART, "dragonheart-pkgs";
for (<DRAGONHEART>) {
        chomp;
        /\/(.+)/;
        $pkgMaint{$1} = "dragonheart";
}

close DRAGONHEART;

open PACKAGES, "pkg-comp-times";
for (<PACKAGES>) {
        chomp;
        /^(.+)\:/;
        $pkgMaint{$1} = "unknown" unless (exists $pkgMaint{$1});
}

close PACKAGES;

open WRITE, ">maintainers";
select WRITE;
for (keys %pkgMaint) {
        print "$_: $pkgMaint{$_}\n";
}
```

C.3.3 pkg-comp-times-retrieve

```
#!/usr/bin/perl -w
#
use strict;
use vars qw [ %completedMerges ];

open EMERGELOG, "/in/emerge.log";

my @TEMPLOG;
while (<EMERGELOG>) {
        push @TEMPLOG, $_;
}

for (reverse @TEMPLOG) {
        chomp;
        if (/\+/) {
                s/\+/'\+'/g;
                #$_ = eval $_;
        }

        if (/^(\d+).*\:\:\: completed emerge.*\/(.*)-\d/) {
                $completedMerges{$2} = $1;
                next;
        }

        if (/^(\d+).*\>\>\> emerge .*\/.+/) {
                for my $pkg (keys %completedMerges) {
                        next if ($completedMerges{$pkg} =~ /\d+ \d+/);
                        if (/^(\d+).*\>\>\> emerge .*\/$pkg/) {
                                $completedMerges{$pkg} = "$completedMerges{$pkg} $1";
                        }
                }
        }

        if ($_ eq "") {
                next;
        }
}

for (keys %completedMerges) {
        $completedMerges{$_} =~ s/ /\-/;
        $completedMerges{$_} = eval $completedMerges{$_};
        print "$_: $completedMerges{$_} \n";
}
```

C.3.4 retrieve-pkg-commit-times

```
#!/usr/bin/perl

use strict;
use HTTP::Date;

use Memoize;
memoize('sampleMean');

my @X;
# We want to capture the following data: The time of every single version bump and bug report.
my @verBumps;
my @bugBumps;

my $pkgNum = 0;

for my $cat (glob "/in/gentoo-cvs/gentoo-x86/*") {
       next if ($cat =~ /CVS$/);
       next if ($cat =~ /eclass/);
       next if ($cat =~ /licenses/);
       next if ($cat =~ /profiles/);
       next if ($cat =~ /scripts/);
       for my $pkg (glob "$cat/*") {
               next if ($pkg =~ /CVS$/);
               next if ($pkg =~ /metadata\.xml/);
               open CHANGELOG, "$pkg/ChangeLog" or die "Unable to open $pkg/ChangeLog: $@";
               while (<CHANGELOG>) {
                       chomp;
                       if (/^\*.+-\d.*-r\d+ \((.+)\)$/) {
                               my $time = str2time($1);
                               if ($time <= 0 || $time > 1163894653) {
                                       warn "Ugh!: $pkg: $_: $time";
                               } else {
                                       push @bugBumps, $time;
                               }
                       } elsif (/^\*.+-\d.* \((.+)\)$/) {
                               my $time = str2time($1) or next;
                               if ($time < 943142653 || $time > 1163894653) {
                                       warn "Ugh!: $pkg: $_: $time";
                               } else {
                                       push @verBumps, $time;
                               }
                       } else {
                               #print "Case not caught at $pkg\n
                               #       line $_\n";
                       }
               }
               $pkgNum = $pkgNum + 1;
       }
}

print "Total number of packages: $pkgNum\n";

@verBumps = sort { $a <=> $b } @verBumps;
@bugBumps = sort { $a <=> $b } @bugBumps;

for (0..@verBumps - 1) {
       push @X, $verBumps[$_ + 1] -  $verBumps[$_];
}

pop @X;

my $total = 0;
for (@X) {
        $total = $total + $_;
}

my $sampMean = $total / @X;

$total = 0;
for (@X) {
       $total = $total + ($_ - $sampMean)**2;
}

my $sampVar = $total / (@X - 1);

my $avgVerBumpTimes = ($verBumps[-1] - $verBumps[0]) / (@verBumps - 1);
print "Total version bumps: ". @verBumps ."\n";
print "Average time between a version bump: $avgVerBumpTimes\n";
print "Sample mean for version bumps: $sampMean\n";
print "Sample variance for version bumps: $sampVar\n";

@X = undef;

for (0..@bugBumps - 1) {
        push @X, $bugBumps[$_ + 1] -  $bugBumps[$_];
}

pop @X;

$total = 0;
```

```
for (@X) {
        $total = $total + $_;
}

$sampMean = $total / @X;

$total = 0;
for (@X) {
        $total = $total + ($_ - $sampMean)**2;
}

$sampVar = $total / (@X - 1);


my $avgBugBumpTimes = ($bugBumps[-1] - $bugBumps[0]) / (@bugBumps - 1);
print "Total bug bumps: ". @bugBumps ."\n";
print "Average time between a bug bump: $avgBugBumpTimes\n";
print "Sample mean for bug bumps: $sampMean\n";
print "Sample variance for bug bumps: $sampVar\n";

sub sampleMean {
        my $n = shift;
        if ($n == 0) {
                return 0;
        }
        return (&sampleMean($n - 1) + (($X[$n] - &sampleMean($n - 1)) / $n));
}

sub sampleVariance {
        my $n = shift;
        if ($n == 1) {
                return 0;
        }
        return (((1-(1/($n-1))) * &sampleVariance($n - 1)) + ($n * (&sampleMean($n)-&sampleMean($n - 1))**2));
}
```