

Table of Contents

System Utility Programming	1
<u>Jason R. Fink</u>	1
0.1.1 Preface	2
0.1.2 Who Should Read This Book?	3
0.1.3 How to Use This Book	4
0.1.4 Related References	5
0.1.5 Conventions Used	6
0.1.6 Acknowledgements	7
1.0.0 The New Programmer's Road Map	8
<u>How the Road Map Will Be Structured</u>	8
<u>How to Get the Most from this Series</u>	8
<u>UNIX Programming Ideals</u>	8
<u>General Programming Ideals</u>	9
1.0.1 Choosing an Editor and a Programming Primer	10
<u>A Primer for Programming</u>	10
<u>Wrap Up</u>	11
1.0.2 Scripting Languages	12
<u>Python, the New Intro Language</u>	12
<u>Shell Scripting</u>	12
<u>Final Notes</u>	13
1.0.3 More Programming Languages	14
<u>Perl</u>	14
<u>JAVA</u>	14
<u>C/C++</u>	14
1.0.4 GUI Languages	16
<u>Scripting/Programming</u>	16
<u>tcl/tk</u>	16
<u>GTK+</u>	16
<u>Others</u>	16
1.0.5 Reflections	18
<u>The Case of Matching A Goal</u>	18
<u>The Case of Playing Nice</u>	18
<u>Applying All Examples</u>	19
<u>No Program Is A Dumb Program</u>	19
<u>Learning To Complement Your Skills</u>	19
<u>The Road Map Is A Guide, Not A Rule Book</u>	19

Table of Contents

1.0.5 Reflections

<u>The Final Recommendations</u>	20
<u>Books - and Lots of Them</u>	20
<u>On Line Data Mining</u>	20

1.1.0.i Facets of Open Source: Defining a Process.....22

<u>What is the Cost?</u>	22
<u>Compiler Toolchain</u>	23
<u>Database Products</u>	23
<u>Operating Systems</u>	23

1.1.0.ii Facets of Open Source: Examples.....25

<u>Open Source Models</u>	25
<u>The "Problem" with the Model</u>	25
<u>Mitigating the Time Problem</u>	25
<u>So How Does the Open Source Model Differ?</u>	26
<u>3 Real World Examples</u>	26
<u>The Scenario</u>	26
<u>The Solution</u>	26
<u>The Resultant</u>	27
<u>The Scenario</u>	27
<u>The Solution</u>	27
<u>The Cost</u>	27
<u>The Resultant</u>	27
<u>The Scenario</u>	27
<u>The Solution</u>	28
<u>The Cost</u>	28
<u>The Resultant</u>	28
<u>Summary</u>	28

1.2.0 Syntax and Context Grammar and Programming.....29

2.0.0 The Bash Shell.....32

2.0.1 Relatively Small Bash Scripts.....33

<u>2.0.1.i. Otop</u>	33
<u>2.0.1.ii MD5 Script</u>	33

2.0.2 Bash and I/O.....37

<u>2.0.2.i. Finding Header Files</u>	37
--	----

2.0.3 Bash Network and OS Scripts.....42

<u>2.0.3.i Remote Sync Script</u>	42
<u>2.0.3.ii Service Script</u>	43

2.0.4 Bash Program: cnchk.....46

Table of Contents

<u>2.0.5 Bash Program: vmware init</u>	52
<u>Config File</u>	52
<u>The Full Script</u>	54
<u>2.1.0 The Perl Programming Language</u>	57
<u>2.1.1 Small Perl Script Examples</u>	58
<u>2.1.1.i Nuke M's</u>	58
<u>2.1.1.ii Return the Contents of File</u>	58
<u>2.1.2 File I/O and Filesystems</u>	59
<u>2.1.2.i Group Report</u>	59
<u>2.1.2.ii Logging Stop and Restart Functions</u>	60
<u>2.1.3 Network and OS</u>	61
<u>2.1.3.i IP Connection Tracker</u>	61
<u>2.1.3.ii Creating A Daemon Process With Perl</u>	62
<u>2.1.4 Perl Program: Host Watch Daemon</u>	65
<u>2.1.5 Perl Program: Nagios Check System Health</u>	68
<u>2.2.0 The C Programming Language</u>	73
<u>2.2.1 Small C Programs</u>	74
<u>2.2.1.i Print User Information</u>	74
<u>2.2.1.ii etu v 0.0.1</u>	76
<u>2.2.2 File and I/O Examples in C</u>	77
<u>2.2.2.i lscpu</u>	77
<u>2.2.2.ii mmw</u>	80
<u>2.2.3 Networking and OS C Programs</u>	86
<u>2.2.3.i Making Forks</u>	86
<u>2.2.3.ii A Tiny Packet Sniffer</u>	87
<u>2.2.4 C Program: The Enlightenment Thumbnailing Utility</u>	89
<u>2.2.5 C Program: Network Decoder - ndecode</u>	97
<u>A.0.0 Miscellaneous Topics</u>	101
<u>A.0.1 Options Parsing</u>	102
<u>use Getopt::Std</u>	105
<u>A.0.3 The Basics of Make</u>	109

Table of Contents

<u>A.0.4 Local Perl Libs</u>	110
<u>The Subroutines</u>	111
<u>Documentation</u>	112
<u>Summary</u>	114
<u>A.0.5 Return Values</u>	115
<u>Does this Make void and no-ops bad?</u>	115
<u>Case Study: Checker Shell Script</u>	115
<u>In C Please</u>	116
<u>How and When is too Much?</u>	117
<u>Summary</u>	118
<u>A.0.6 Goofy Snippets</u>	119
<u>A.0.6.i Inline Bash Function</u>	119
<u>A.0.6.ii Tiny Inline ASM in C</u>	119
<u>A.0.6.iii Tiny Inline C in Perl</u>	119
<u>A.0.6.iv ISBN Processing Using LISP</u>	120
<u>A.0.6.v Elementry Congruence in Python</u>	120
<u>A.0.6.vi NASM Boot Floppy</u>	121

System Utility Programming

Jason R. Fink

(c) Copyright 2010 All Rights Reserved.

0.1.1 Preface

System utility programming can mean a lot of things. In the context of this book utility programming means writing scripts and programs that interact at a very low level with an Operating System. Further defined: shell and compiled programs that will run in the UNIX shell environment . That is not to say such programs cannot communicate with higher level software - in fact well written tools that reveal data about system details rely heavily upon low level utilities to collect the information for them and report it back. The very essence of a well written utility is that it provides easy to read data that can be used by virtually any other utility or piece of software on the system.

The Operating Systems used in this book are Linux kernel based or BSD derived flavors of UNIX. UNIX-like systems are used for several reasons:

- The author is most familiar with them
- They are free of charge to download and install
- They are the most well equipped systems for the examples

Of course bullet 3 and 1 are tied but it is true; UNIX and UNIX-like systems have been around since their epoch (1969) for a reason: they have a strong foundation and rich development environment. It is more difficult these days to install a system without Perl for example then the reverse. Most free Operating Systems come pre-loaded with compilers, toolchains, interpreters of all sorts and a variety of shells.

In this book system programming will be looked at from a system administrator's point of view; much of what is written hinges on getting information about the system in a format the user wanted. That said none of the programs are especially large (only a few go beyond one or two pages). This book is designed to help lay a rounded foundation for the potential future programmer and not meant to be an end all resource or even a practical reference manual.

This book is released under the Creative Commons License version 3. Per the license text (which can be found at: <http://creativecommons.org/licenses/by-nc-nd/3.0/>):

You are free to copy, distribute and transmit the work under the following conditions:

- Attribution: You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- Noncommercial: You may not use this work for commercial purposes.
- No Derivative Works: You may not alter, transform, or build upon this work.

With the understanding that:

- Waiver: Any of the above conditions can be waived if you get permission from the copyright holder.
- Public Domain: Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.
- Other Rights: In no way are any of the following rights affected by the license:
 - ◆ Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
 - ◆ The author's moral rights;
 - ◆ Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.

0.1.2 Who Should Read This Book?

Anyone new to the shell scripting or programming world will likely find some useful information here. System administrators looking to expand their skills or perhaps leverage programming to their advantage. System programmers who work on non-UNIX systems would likely find something interesting as well - in the least perhaps a style contrast.

A solid understanding of basic scripting (shell or Perl) and C programming is required. Remember, this book gives examples in several languages but it will not stop to explain every aspect of each language used: there are plenty of other references for that sort of thing.

The basic skillset for this book is:

- be able to write simple shell and/or perl scripts up to managing arrays (or in some shells just handling lists).
- be able to understand simple C code (up to basic data structures and pointers)

0.1.3 How to Use This Book

The examples in the book are just that: real world examples the author had to write for one reason or another. In several of the examples it should be obvious that there is a better (and faster) way to do things. The `lscpu` program comes to mind; a program written in C which reads a variety of data from the Linux `/proc` filesystem should have simply been a shell script that greps and awks out what it needed, however, it makes for a good example of how to leverage the file I/O capabilities of C and at the time it seemed like a good idea.

The primary motivation of this book is provide the reader with small compact examples they can use as a springboard for their own endeavors. Readers would be advised to also have available programming references (either online or printed) for some of the trickier corners navigated within the texts.

0.1.4 Related References

Following is an evolving list of excellent programming books:

- Art of UNIX Programming <http://catb.org/esr/writings/taoup/>
- Perl for System Administration <http://oreilly.com/catalog/9781565926097/>
- Practical C Programming <http://oreilly.com/catalog/9781565923065/>
- BASH Programming HOWTO <http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html>
- Wicked Cool Shell Scripts <http://nostarch.com/wcss.htm>
- C Programming FAQ <http://c-faq.com/>

0.1.5 Conventions Used

I decided to keep the format as simple as possible in the book. Any code or command lines are simply indented in their own textarea like so:

```
prompt$  
prompt$ wc -l /etc/passwd  
prompt$
```

Which includes single line commands or code listings. Code listings and commands are separated by context - it will be very obvious to the reader when it is a command or code. For example in this faked up excerpt:

```
int x;  
int y;  
char * foo;  
char * bar;
```

0.1.6 Acknowledgements

First and foremost every and anyone involved in Free Software and/or Open Source. Without all of our efforts the capability to even do most of what is in this book (and a great deal of others) would never have been realized.

Secondly to my family friends for motivating me to work just a little every night.

Thirdly by name some other Authors:

- Marshall Kirk McKusick
- Eric Steven Raymond
- Paul Graham

1.0.0 The New Programmer's Road Map

There is nothing I love more than when someone says "You do it this way" then gets up and walks away. This statement rides right along side of the nefarious RTFM - somewhat true of learning how to program. Many people will simply say do this, that and the other

The goal of this series of texts will be quite simple: To provide information about the most popularly accepted method(s) of learning how to program.

What this series will not do is instantly turn you into a programmer in "21 Days" - or whatever. Learning how to program takes a lifetime because programmers never stop learning. Whether it be new languages, styles or techniques - the process goes on forever.

How the Road Map Will Be Structured

By no means do I intend on writing a set of primers, tutorials (there are plenty of these already) or cast my own distorted views, instead the Road Map will approach programming from a high level perspective. It will introduce what many believe are the best ways to get started and where to go from there. The series will also look at attending classes, purchasing books and other ways to become a proficient programmer.

The beginning of the series will be a forecast of what is to come and some general philosophy ideas, after that we will look at editors and ways to "Prime up to Programming". Next will come some suggested starter languages, middle languages and finally a sort of "pick your own way" after that.

It is important to note here that it takes all kinds, which is why this Road Map will be somewhat vague or even appear to be indecisive, it is a guide, not a rule book. Many people want to program in different ways for different reasons. For instance, take the differences between people who may spend a lot of time working on GUIs versus Network Programmers. Many programmers have a particular area they like (this seems especially true in Free and Open Source Software Arenas) to work in, the Road Map will attempt to illustrate these differences instead of saying "Learn this and you will forever be employed."

How to Get the Most from this Series

This series will not be designed so easily to where you could go learn every primer, read every paper and perhaps do everything mentioned in one text and go to the next. Instead it is a series of texts on progression. The best way to use the series would be to do as much as you can, move at your leisure and refer back to areas you may wish to reread.

UNIX Programming Ideals

I am not one for duplication, however, I will restate some very popular beliefs about UNIX Programming that have been around since before UNIX was around. I would give due credit to Dennis, Ritchie and Kernigan (in addition to the scores of scientists before them) plus Eric Raymond for bringing their ideas together in many of his texts and of course RMS with his brilliant coding contributions. Following is a short list (not all of them) of some of these ideals:

- Small: always try to keep it small
- Clean: clean code makes for scalability

- **Connectivity:** by this I do not mean networking, instead I refer to connecting pieces of software together.
- **Re-usability:** One of the most important features of good UNIX programming - never do the same thing twice.
- **Proficient Efficiency:** writing efficient code that is understandable. A difficult endeavor.

These ideals (and more) are expressed much better in ESR's *The Art of UNIX Programming*; Chapter 1; *Philosophy Matters* (a work in progress). I highly recommend reading it.

General Programming Ideals

Just about every really good programmer will tell you (and I was told the same thing) you should learn a variety of different types of languages. There is a reason for this beyond the obvious job security (although it helps). Different programming languages, literally, require different ways of thinking. I have assembled a short list of language categories, by no means is it authoritative as much as it is descriptive of the types:

- **Machine Level:** Extremely low level code that interacts directly with or "one step above" the hardware.
- **Precompiled:** A file (or set of files) is fed to a compiler and executable (object) code results.
- **Interpreted:** A file with syntax is fed to an interpreter which reads the code and executes (in some cases the code is compiled into executable code and then executed).
- **Scripting:** Scripting can often be done as the Interpreted type (stored in a file then sent to the interpreter) or the interpreter can be started interactively and code can be written on the fly.
- **Highly Objective:** Yes I made that up. This might be languages that are built around being object oriented.
- **Other:** Here languages such as Virtual Machine types, LISP Processing and perhaps low level database programming lie. In other words, anything I have yet to classify or was too lazy to think about.

An important note here is that different problems do in fact require different languages. What may seem like a practical solution in one language may be extremely difficult or even prohibitive in another. For instance, let's say we are using an interpreted language which cannot produce binaries you can keep on your local system. You have solved a simple problem with this language, however, the code is only a couple of lines. Additionally, your interpreter is 12MB in size. So when these couple of lines are sent, all 12MB of the interpreter are loaded into memory. Perhaps it would be better to write the program in a language that produces a small binary, hence saving some overhead or even using a different interpreter that is smaller. This is the essence of the right tool for the right job. Many large projects use a variety of languages to piece together one large functional product.

Another important ideal is that one does not have to become proficient in every type of language out there, I do believe that would be quite difficult. Instead, learn a variety of types to the degree where you can do something with them (even if it is trivial) so you gain an appreciation for the design, and even perhaps see a way to attack a problem with a language that you already are proficient in, again, brain exercising and perspective.

1.0.1 Choosing an Editor and a Programming Primer

There is a silent holy war (among many thousands to be sure) on the Internet. It can be found waging in newsgroups, chat rooms, "web forums" and the like. It is about the Editor of Choice. Some will say it is vi while others will insist the end all editor (and environment in some cases) is emacs. I would say - the choice is yours, but what value might I bring you without some simple information about them first?

Initially built by Bill Joy of Sun Microsystems, the vi editor represents simplicity and elegance. It has a wide range of commands and can have MACROS. The vi editor is best known for being light. It normally has only a terminal interface, however, there are clones out there that can run in X mode. A very thorough tutorial on the vi editor can be found at the Engineering Computing Network of Purdue University's VI TUTORIAL.

On the other hand is emacs from Richard M. Stallman, founder of the GNU project. Emacs has an intense array of commands, extensions and tools to help a programmer a long. In fact, emacs is similar in nature to an Integrated Development Environment, but better. One can also use mail utilities, play games, download and participate in newsgroups with emacs. The trade off for some might be the time it takes to reap the full benefit of all the things emacs can do.

There are also many other editors for UNIX out there, some are like vi, some are like emacs while others are sort of hybrids. If neither one of the ones mentioned here suit your tastes, explore and see what else is available, I am sure you can find one that works for you.

So which one do I use? I prefer the vi editor, mainly because I am on so many different platforms and I will be blunt, I am lazy. It was easy for me to learn and retain the commands. I used xemacs for awhile but was easily frustrated by my own poor memory while trying to learn how to use all of it. I have always been a sucker for the whole simple elegance thing too.

A Primer for Programming

I had intended to suggest learning either the Hypertext Markup Language (HTML) or a Text Processing Language as a "primer for programming" but decided to stick with HTML only for several reasons. First, it is somewhat difficult to learn a text processing language and the processing side as well. Second, many people suggest learning HTML before a text processing language - that sort of ended the issue.

Now, what do I mean by Primer for Programming? Well, just that, through HTML, one can learn the basics of structure, a very important aspect of programming. I do not believe one needs to become a master of HTML, rather the opposite, understand how it works and what happens when one uses certain tags. In a sense it is a Primer for Programming Primers if you want to get technical.

HTML is a subset of the Standard General(ized) Markup Language (SGML) which is now coming "online" as the eXtensible Markup Language (XML). HTML is a set of tags that are relatively easy to learn and employ, however, it is a Markup Language which means the author is concerned not with appearance but with the Structure. This is why HTML makes a great primer for primers, some (not all) principles of programming can be found within it.

Another great thing about HTML is it can be fun. It offers very quick rewards for a small amount of work, always a good thing for the aspiring programmer since later, things will not be so easy.

If you are strictly interested in programming, I suggest learning basic HTML and going no further in the "on line" endeavors until you have a little spare time. Otherwise, for the extremely brave who wish to know and understand a great deal of how HTML works and additional topics such as style sheets.

Wrap Up

Try out an editor and play with it, once you feel somewhat comfortable with it, start learning HTML using your editor. This will give excellent practice with your editor and a chance to have a little fun with HTML.

1.0.2 Scripting Languages

Python, the New Intro Language

The Python programming language is an interpreted type of language in that source code is not compiled into binary form. It is also the most recommended introductory language by many good sources (ESR among them) for several reasons. I have learned Python all the way up to creating a class and I admit, I liked it a lot. Some of the main reasons that Python is popular as a first language is that it is very forgiving and helpful about syntax errors when not used interactively. If used interactively it can be somewhat difficult. Python also has another interesting facet, it uses indentation to determine how statements are to be operated upon and does not have closing marks for statements, instead multi-line statements are marked like a multi line UNIX command with a backslash.

Another reason that Python is preferred as a starter programming (or scripting) language is that it has a great deal of built in capabilities. For instance, console I/O is extremely easy both ways, making things like I/O loops easy to write. What that means is one can learn some very handy techniques without doing a great deal of coding thus freeing up the learners mind to explore and be creative early on instead of memorizing a great deal of long programming to accomplish such techniques.

Finally, Python scales quite well as a learning tool goes. One can learn it at a basic procedural level and stretch all the way to the fundamentals of Object Oriented Programming (OOP) in a relatively short period of time. As a case in point, I will use myself since I really do believe if I can do it, anyone can. My first language was Perl (I did shell program before that), it took me three months before I could write a Perl program that could do something worthwhile and believe me, it was not object oriented. It was strictly "block level" programming. Next I learned C and C++ (sort of at once) and it took me about three months until I reached and mastered the basics of OOP in C++. With Python it took me one week. Now, granted, I already had the Perl, C and C++ experience, but I still believe that was fast.

So as you can see, Python is very good as a first language for the following reasons:

- It is very forgiving
- It has a great deal of good built ins (and modules)
- It scales very well
- It frees up the learners brain to focus more on techniques

Shell Scripting

Shell scripting is not just a great way to get a feel for programming but is an essential skill for UNIX power users (e.g programmers). Understanding shell scripting and the shell environment (whichever one you choose) is essential. I can guarantee it will come in handy someday. Shell scripting is basically putting UNIX commands in a file along with shell built in functions to do things (such as if iterations for example). Obviously you can already see the inherent advantage to learning it. I use shell scripts for maintaining file systems, log-files, sources and my own packaging. I also use them to rapidize packaging for large projects until I am ready to finish the Makefiles associated with them. In addition to the obvious benefits of learning shell scripting, there is also it's similarity to more procedural types of languages.

Shell scripting (and programming) is also a vague term because, there are many to choose from. Following is a short list of the ones I have seen (not necessarily scripted in):

- sh - bourne shell
- bash - bourne again shell
- ksh - korn shell
- csh - cshell
- tcsh - an enhanced cshell
- zsh - a sort of combination shell
- ash - minimalist shell
- sh-posix - the "posix compilant" shell

Final Notes

There are other languages out there for starting, however I honestly believe Python is the best starter language. Some readers recommended ADA or Pascal. I definitely do not agree with Pascal as a starter language - or for anything actually and I have never used ADA so naturally - I cannot recommend it either.

1.0.3 More Programming Languages

Without out a doubt the three big (more like three and a half) languages out there today are C/C++, JAVA and Perl.

Perl

Perl is an easy to learn, easy to write language that is great at operating on strings of arbitrary length(s). As such the Perl acronym of "Practical Extraction and Reporting Language" has been comically portrayed as "Practically Eclectic Rubbish Lister" or reduced to "awk on steroids." While both may have a ring of truth to them, they do not do Perl justice, not by a longshot.

So what makes Perl so great? Time. As I mentioned, it is arguably as easy to learn as Python, however, it does not have a lot of built in pragmas one should follow. As such sometimes someone else's Perl code can be a bit hard to read.

Systems Administrators (myself included) use Perl every day (either automatically or not) to compile reports, trim/filter logfiles, run filescans and a plethora of other chores that used to be done by shell scripts. Perl is also interpreted like Python, so there is no pre compiling done, well, that is not true exactly. The interpreter reads the source file, compiles the executable and then runs it. In a sense Perl more or less cuts out the middle (make) man. Systems Administrators (myself included) use Perl every day (either automatically or not) to compile reports, trim/filter logfiles, run filescans and a plethora of other chores that used to be done by shell scripts.

Perl is also robust enough to perform applications programming, especially web applications where standard input is involved - again - string manipulation. One of the great things about Perl is scalability. Not unlike Python, Perl inherently scales rather well from scripting language to all out applications language.

JAVA

I have to confess I do not know a lot about JAVA. I wrote one of the many famous calculator programs and lost interest. What I can say about it is I am always surprised at the variety of uses and places JAVA seems to be popping up and, in reality, has always been since it popped up on my geek radar. Initially I thought JAVA was another GUI language, the difference between it and other ones being it has the "write once run anywhere" action going.

JAVA has turned up in a number of other places as well. JAVA is used to power appliances and micro devices of all sorts. On fast intranets it makes a great user interface tool that scales rather well. JAVA in addition to many other great languages exudes this ability to scale.

C/C++

Definitely different languages, but evolved from the same core, C and C++ are the most widely used programming languages to date. The reason is simple, while they may not be "write once run anywhere" like JAVA - they came closer to it sooner than any other programming language for its time. Especially C which was more or less designed to be easily ported.

Still, many programs from many languages can be ported. Another little item about C in particular that sets it apart is it's easy access to hardware through small, clean interfaces. Hence why so many systems level

programs and utilities are written in C or C++. Due to the low to high level nature of C/C++, almost all applications that are multi platform are written in C/C++ (including - other languages).

Last, and definitely not least, resulting executable object code is tiny. C/C++ (again, esp. C) are great for writing small, clean and efficient programs. They are the language of choice for device drivers, utilities and low level programs for that very reason.

So what, prey tell, is the downside to C/C++? Well, I have heard this a million times so I will repeat it;

"With C, we have the ability to write great programs and shoot ourselves in the foot in the process."

" . . . with C++ now we can blow off the whole &^%\$ leg."

That was paraphrasing a bit but true. It is easy to cause buffer overflows (without even knowing it) or the most famous, hidden errors. I believe logical errors can be done easily in any language but I have to admit they seem somewhat more prevalent in C/C++. I would also venture to say there are so many problems in C/C++ programs because of wide use.

1.0.4 GUI Languages

Scripting/Programming

One thing I never expected, being such a geek and all, was to actually like creating a Graphical User Interface of any sort. The reason I might not like it is quite simple, I never really used them a lot (with the exception of running X windows so I can have, lots of xterminals). It occurred late one night when I was bored out of my mind. I picked up some unleashed book that was laying on the floor and just thumbed through it to find something interesting.

I stumbled across an introduction to tcl/tk GUI scripting and tinkered with it for a bit, I was very surprised . . .

tcl/tk

Tcl/Tk was fun, I really like the interactive scripting side of it, which I can tell would be difficult for a large project. In case you are wondering exactly what the heck I am talking about, please allow me to explain.

Tcl/Tk has an interactive mode (not unlike Python or LISP Processing for example). You can build objects just by typing in the syntax in interactive mode. So in effect, I would type in a bunch of syntax and suddenly off somewhere on my X display a little box pops up; then a button, a label, some more buttons, more labels etc. Eventually I made it do something (I honestly cannot remember now what that something was - I am sure it was cool though).

At the time, I thought this was just too nifty and really, still do. It was a heck of a lot of fun and I saw why tcl/tk had become so popular. Not unlike other GUI languages, a lot of people found it a blast to hack on and shared the results (whether useful or not).

GTK+

The other popular GUI language I am going to discuss is GTK+. While GTK does not offer the interpretive interface that Tcl/Tk does, it is still quite rewarding: anyone who has worked even with interpretive languages realizes you must enter all of the syntax in a file anyway on larger projects. So making the change was not exactly difficult. The reason I tried out GTK+ is simply because, well, I had to or else I could not possibly write about it, could I?

GTK+ uses a C-style syntax where most of what one does in the way of generic operations is done by using the libraries. To me, this is no different than doing my homework with the glibc documentation, and for this I see why GTK+ is so popular. A C or C++ programmer should have no problems (after a bit of reading) getting up to speed with GTK+. Since I have a preference for C/C++ of course I really enjoyed it and definitely recommend it for a C/C++ programmer on UNIX.

Others

Are there other programming languages? Heck yes. Thousands, literally. Not only are there thousands, there are variants of existing languages. So what among these thousands, would I recommend learning? Well, for starters, I strongly recommend learning one of the LISP Processing (or LISP if you will) languages to the point of creating a recursive function for one simple reason, LISP uses recursion by default; once you understand this you do not really have a need to go further, but if you wish to, please do so. The sheer elegance of LISP,

aside from it's more well known acronym of:

Little Irritating Special Parenthesis

is truly an eye opener--bluntly, it is a beautiful language and I liken learning it (even at a basic level) to learning a new human language.

1.0.5 Reflections

One point I did fail to reiterate in subsequent texts that I made in the first text is that I do not believe and anyone in the right mind would also, that any one single language is the end all solution. I do not argue that there are some languages out there that are bordering on the edge of pure blasphemy. It has always been the case for me that it truly depends on what the goal of a particular project is or where it will fit in with other pieces of what it may be working with. That may be somewhat confusing so let us take a look at some quick examples to further illustrate those ideas:

The Case of Matching A Goal

In this example, we have a project that is not necessarily connected to anything else, so we have a lot of options, we could, literally, choose any language if we knew the user of the resulting application or utility would have everything they need to use it. In this case, we look for the best match to meet the goal. If we wanted to create a very fast small utility, then C, C++ or Perl might be the best language to use, if we wanted the ultimate in speed, I would venture to say C or ASM (or a combination therein) would be the best match.

If the goal was slightly different and I mean just slightly, we might have to shift to a different language. What if we wanted a strong object oriented program, C++ or python might be the answer. If we wanted a great deal of speed but to still retain the built in Object Oriented features then C++ might be the best match for that particular goal.

Finally, for the sake of argument, let us assume that we will be doing massive string handling on a pretty powerful system. Perl would be my first choice, at least as a prototyping language for it's ease of auto-magically handling arbitrary string lengths.

The Case of Playing Nice

There are occasions when one should or even have to use a certain language. If we are creating a GNOME application for example, it would indeed make sense to use GTK and it's associated tools to build our application. Another time we would be forced to use a certain language would be kernel programming, where C and ASM play a large role.

Unfortunately, it is rarely ever so clear cut. I can speak from personal experience about this because I face it for every project I start. In my case it is almost always a sysadmin tool of some sort, well except for my nefarious screw off programs. So often I must think about how I want to approach a problem and how it would work the best but still be created with a language that other toolmakers use. Now, why should I care? It is Free/Open Source, which means if there is a problem that exists and I miss it (and believe me, I do) then there is a good chance that if someone else wanted to fix it or make a change for the better they can because they will be familiar with the language.

In my case I choose C or Perl almost everytime. I primarily use C when working very close to the kernel or hardware. When building admin tools like quasi greppers, log filters and report generating scripts I almost exclusively use Perl. Both languages are extremely popular (if not necessary) to the audience I happen to cater to.

Applying All Examples

In both of the aforementioned cases and examples, everything goes when it comes to selecting a language and even the implementation, coding style, commenting style - whatever. Try to match every aspect of the project with:

- The Goals
- The Environment
- The Possible Contributors

Following those guidelines will tend to make life a little easier.

No Program Is A Dumb Program

It might be a poorly written one, but the intent is always good. The first time I wrote a real program (although I would hardly call it that now) it was not exactly an achievement in programming. It worked, don't get me wrong, it did the job, but it could have been a lot better. Regardless, it was a learning experience. That is what it is all about really; learning, growing and expanding.

A case in point (again), I made what might be considered one of the single most useless programs of all time. It is written in Common LISP Processing (CLISP) and contains a whopping 12 lines of code. I read the entire CLISP tutor and implemented every example and when I finally understood how it worked, I decided to write a recursive CLISP program, my motivation was irony. You see, LISP uses built in recursion in its statements. So, I thought it would be somewhat ironic if I wrote a useless program that was a call to one recursive function. The program runs through ninety nine bottles of beer on the wall, almost instantly the screen is filled with lines of the song.

Hey, at least it is efficient.

Point is, I did it as a learning experience and shared because I felt others might get a kick out of it. I did intend to improve upon it, but I couldn't stop laughing after the "first revision."

Learning To Complement Your Skills

One last shot I will fire at the "learn a variety" ideal is understanding the approaches of one language and using them in another. In some cases there is a direct translation. Say you know C++ but not C and are programming in C. You stumble across a linkage method and realize you can do the exact same thing in C++. You have just added to your armory of C++ skills.

Of course, that is a pretty cheap example since C++ is an enhanced C with built in Object Oriented capabilities. Instead look at how CLISP promotes the use of recursion (since internally that is how it works anyway). Understanding this can help one better understand how to use recursion efficiently in other languages or to create built in lexicals using recursion in another language.

The Road Map Is A Guide, Not A Rule Book

Experienced Role Playing Gamers will remember the cardinal rules set out by Gary Gygax in Advanced Dungeons and Dragons when he stated (and I am paraphrasing):

"These books are not rules, they are guides. They show a path but not an absolute way to do things."

As is the New Programmer's Road Map. If at some juncture one decides that using a language I did not even mention (and I did not mention a lot) would suit them best then so be it. Instead the New Programmer's Road Map is a really big suggestion. I believe following it to some extent will round out ones skills, but that does not necessarily mean that a New Programmer should follow each and every suggestion I make. Instead, I would make these ones as a sort of short version:

1. Prime yourself by choosing an editor and fooling around. Possibly make some sort of Markup Document(s) with it.
2. Choose and easy to learn language.
3. Move onto a middle level or lower level language
4. Next choose either an Object Oriented language, progress a current language you know into OOP or try a GUI language
5. Try to find a language that seems altogether different to you, not just in syntax but style and philosophy as well
6. Home in on a couple of languages you liked

In my case, it was quite typical of a sysadmin career really. Here is the short short version of what I learned and in what order:

1. Tinkered with JAVA
2. Shell Scripting
3. Shell Programming
4. Perl Scripting and Programming
5. C and C++
6. a little Python, Tcl/tk and GTK+
7. Tinkered with CLISP

I focused on Perl and C for the most part. As you can see, I sort of do not follow the Road Map exactly, hence why I thought I might share my thoughts with the world at large!

The Final Recommendations

The Road Map missed a few points from the beginning, the most important one not covered in this installment is - what else do I need?.

Books - and Lots of Them

Buy books, lots of books. You will need them. I cannot really recommend one brand over the other, I have several books from several publishers on C, C++ and Perl (and soon to have Python). All of them have different examples, different approaches and methods. After a time you will even begin to question the material you bought. This is a good thing. If I did not receive an email that questioned something I said here, I would REALLY be scared. If you do not question, you will not learn . . .

On Line Data Mining

Almost parallel to books is online documentation, either on the internet, info, ps, man or whatever. Dig, Dig, Dig. It is there, somewhere. I have found this to be the best way to get the full advantage of any library or call

available. Understanding the documentation is an art unto itself. Weeding through the diatribe of one individual can be difficult, but the reward is priceless. Finding that obscure capability you never knew existed is always enlightening.

1.1.0.i Facets of Open Source: Defining a Process

Recently I was asked the following question:

" What is the point of differentiating Open Source as an entity versus something that has just always been there?"

In other words; if a company sells services and products, what reason outside of buzzword compliance is there for creating a practice that revolves completely around Open Source?

Good question - the answer is simple at first:

" Open Source is more than just software - it is a way of doing things - not to be confused with "best practices". Open Source is a pragma versus a method. "

It is easy for people who do not live with Open Source to become confused. The larger the distance from Open Source (for instance - persons who are only business oriented) the greater the confusion. This text examines what benefits of Open Source can mean while the next article will use specific real world examples of Open Source saving time and money. What Does it all Mean?

Open Source and indeed Free Software are not necessarily free of cost. It is free in the sense that the code is available, may be modified for local use and redistributed as long as the original license remains intact and is adhered to. In some cases, upstream patches of changes are required. So what does free really mean? Truly free is the definition of "Public Domain" - or "do whatever you want with it." So where is the cost?

Where a product can be bought and brute force implemented, talent can be purchased and used with accuracy. The entire idea of service versus product is key to Open Source. Only a few years ago, most companies looked at software as a boxed product to buy: direct system administrators and database administrators to put the product in place; then let users and/or developers "have at it". The problem of course is that computing systems are organic at some level even if they do not appear to be at a high(er) level. Computing systems evolve; even if the underlying hardware and software use extraordinary mechanisms for release/change control - the evolution cannot be stopped: the reason for this is simple; stop evolving the systems and you will lose. Changing the thought process from blackbox product to strong resources makes sense. Open Source is an enabler of the idea that powerful resources give advantage as most Open Source is is inexpensive (or free) to use, allowing an organization the financial resources to pay for resources.

What is the Cost?

Is it possible that the costs even out? Certainly. The real answer is "it depends". It truly depends upon the type of software and scope. A good example is to compare the difference between three different software system types in an order of magnitude:

- Compiler toolchain.
- Database products.
- Operating System.

Note that the product is not the only item to consider; there are two other costs:

- Local support/development expense.

- Vendor support expense.

Compiler Toolchain

For those who have ever had to deal with building for multiple platforms (or target architectures) from a single platform; powerful crossbuild toolchains are a blessing ... and wrought with danger. Getting a free crossbuild toolchain is easy: GNU has a great one. The problem of course is support. A company known as Cygnus used to offer support for cross compiling - there are likely more. Most proprietary crossbuild systems have a hefty front end fee plus even larger support costs. In this the smallest example; the benefit of Open Source far outweighs the proprietary alternative. For example, the cost can be less if paying for a person who has the skills to master the toolchain versus buying an off the shelf product with some sort of support agreement.

Database Products

Databases have one main factor: scale. A proprietary database product - top of the line - can come with a rather large (on the order of thousands or even tens of thousands of currency) front end fee before licensing and support are even factored in. If a database does not need features like built in database redundancy (outside of the OS) and will not exceed a few million rows in any given table then there is no reason to sink thousands into closed products at all. Some of the open source products are not only catching up with features like built in redundancy and a multitude of multiplexing access methods, in some areas, they are exceeding their closed source counterparts. This makes the support side easy, with no heavy front end cost; database specific support through a vendor for an open source project is trivial. Conversely, if there are features in the proprietary database that are needed - then that is the way it is ...

... just remember; the Open Source equivalent may be available next year ...

Operating Systems

A complete operating system is entirely different because of support models. Some companies will support software on the operating system in addition to the core OS - which muddies the water a bit. For the time being, things like development and database software will be tossed aside within this context to look only at the core Operating System.

Most, not all, proprietary operating systems are tied to a specific set, class or even their own hardware models. The same can be said of open source operating systems - the difference being the former is a complete product line while the latter is something one can deploy with more flexibility on a particular group of hardware platforms; in some cases - a large variety of platforms.

With most supported Open Source operating systems, there is some sort of fee associated with support and subscription. Support and subscription are also tied to proprietary systems - the difference is the cost (Open Source systems are generally- but not always - less) and hardware. With many (not all) closed systems the hardware is either the same company or a very limited set. The offset cost of open source is dealing with multiple hardware vendors and their support structure. It is worth noting that most Unix shops deploy a mix of vendors anyhow - so the old argument of "one vendor for all" is usually the exception.

The operating system is probably the fuzziest area of support because of the multitude of factors involved:

- Subscription fees (patches etc.)

- OS Support Fees.
- Hardware support; singular or multi-vendor?

In the simplest case, general purpose, large (not very large) database, distributed or commodity systems are best served with Open Source. Only exceptions to the rule apply to closed systems. It is worth noting that this is a complete reversal in thinking dating only a few years ago which stated "Open Source is good only for commodity systems."

1.1.0.ii Facets of Open Source: Examples

In the first text a generalized view of how Open Source may be beneficiary to an organization was highlighted: not this time, now a look at development models and three real world examples of leveraging Open Source resources for a particular organization.

Open Source Models

The Open Source model is actually very similar to extreme programming minus the formality. In the simplest terms; the Open Source model is:

- Someone creates a piece of software, an application or software system and makes both the software and source code available.
- Other people use the software and find and/or help fix bugs.
- Other people make improvements.
- Fixes and improvements either return to the author or are propagated some other way (forking, subset versions, feature releases, patches etc.).

On the surface the model does not sound too difficult and it is not for small projects. The Open Source model becomes difficult - ironically - when the software becomes popular. The more popular a particular type of software becomes factored by the software's size creates a management issue on the author(s) part.

The "Problem" with the Model

Essentially the three stages of time management look something like:

1. {I,We} have some spare time to create this software...
2. The software is popular, it is a good thing there is spare time to maintain it...
3. {I,We} have no spare time...

The time management problem is exacerbated by the number of developers controlling the source, for example, the Linux kernel (for a very long time) only had one maintainer of the core source tree; burnout was inevitable.

Mitigating the Time Problem

Other than dropping a project on the floor, there are several ways to mitigate time management problems:

- A central core team who can make sound experience based decisions on code inclusion.
- Gatekeepers for parts of software systems and/or packages.
- ... and the most common: a combination of the above.

The implications of time management are obvious; a real version system needs to be used, someone or some persons have to set aside time to figure out exactly which model to use and execute putting the mechanisms in place.

So How Does the Open Source Model Differ?

Most organizations take three approaches to software system development:

- One or two people do it all...
- Throw a programming team at the problem(s)...
- Both... (where the former takes over for the latter...)

The models of central core and/or gatekeepers work - in fact; they work the best. The Linux kernel experienced a slowing of development when the single gatekeeper issue arose; it was remedied by putting in a set of gatekeepers to handle subsystems. The various BSD projects use a two pronged approach; there are developers who are considered experts for a general area, set of areas, specific modules and package experts with an accompanying technical core group to make final (and sometimes pretty tough) decisions about code inclusion.

It is worth noting that many organizations do end up using the Open Source structure naturally. A team of programmers is assigned to create an application; often they divide into groups of "who is better at certain aspects of the project" and subsequently end up becoming the subject matter experts; at the same time; the more experienced and/or talented individuals naturally assume leadership roles.

3 Real World Examples

These examples are real; the names involved have been omitted for legal reasons however, each case study example happened. Perhaps even of more interest; each case example happened to the author. The examples are given in an order of magnitude from least to most significant.

The Scenario

Security through obscurity is often a very bad method; key word: often. At the time hardly any sites were using IPv6; the provider for this company was not using IPv6 (at least not routing IPv6).

The scenario is simple, two systems in a DMZ needed to communicate using a private connection. The problem? No physical private connection. Secure Sockets Layer and Secure Shell both can do the job with an encrypted session - what if that could be taken a step further? What if communication was done between the two systems on a non routed network using IPv6?

The Solution

The systems were one NetBSD and one FreeBSD system. They both supported IPv6 by default [1]. Instead of having to remember the syntax of remote execution over IPv6 a set of wrapper scripts was written to use IPv6 in place of the commands needed to maintain a heartbeat between the servers. The Cost

Solely hours; roughly five hours total versus:

- Buying two network cards.
- Installing the cards.
- Configuring the private network.

The Resultant

The organization saved money in time and hardware expense by paying for the technical resource to implement a solution.

The Scenario

Some cases are just too interesting to pass up. A company wanted to use Oracle on Linux in 2000. The problem.. Oracle kept wedging because it was using a newer memory model than the current kernel. The short term solution was to update the 2.4 kernel to at least 2.4.12. Easy enough, the sysadmin downloaded the Red Hat sources; quick built (that is - just use the default) and voila - a new kernel . . . one minor problem . . . the BroadCom ethernet card did not work. It gets worse; the card was not supported at the time nor any of the subsequent kernels [2]. Quite a place to be cornered into. One of the reasons this system had been purchased was for the 1 Gig card.

The Solution

The sysadmin understood that at the time, a new 1 Gig card would cost a great deal of money; possibly thousands of dollars. The solution was simple enough: port the driver from the previous kernel. The admin copied over the driver files, changed the PCI handshaking and glued the card into the newer kernel's framework . . . it worked.

The Cost

Time, however, in this case the time was a direct proportion to the cost of a new card. A new card could have cost at least 1000 USD - the admin (who already had driver experience on another platform) did the work in 2 hours. Remember, the porting was done in 2 hours versus the time to:

- Find a new card.
- Order the card.
- Install and configure the new card.

The Resultant

It is interesting to note that the result was staggering, the system was made available again within two hours of diagnosing the problem whereas if the driver could not have been ported for lack of experience, the cost would have been far greater (possibly an order of magnitude).

The Scenario

In 1999 Brett Lynn put forth the idea of using a method to check all executables in a system (and scripts) via a kernel loaded hash table; not unlike tripwire but inside the kernel. A Fellow Developer felt it was a good idea and decided to take Brett's proof of concept code and break it out into a separate module within the kernel. The same admin took the idea to their corporate management with a singular pitch . . . credit.

The Solution

Not only did the company the sysadmin worked for go for it - they offered up test hardware that both the admin and developers could login to remotely. Eventually the first draft was hammered out and committed to the NetBSD kernel.

The Cost

In this case, the company paid for power and a system plus spare developer time that the admin could have been using to code for the company, however, the mandate was for the admin, no coding unless it is idle time - effectively the company lost no personnel cost.

The Resultant

The company got three things:

- Free advertising in the kernel.
- First use of the technology.
- A very happy employee.

Summary

Upon closer examination it can be found that Open Source differs from closed source only in the sense that vendor support models are modified and may (read: often do) take a little time to adjust. The Open Source development model while on the surface looks like chaos in action is not. Small projects remain controlled by their respective developer(s) while larger ones adapt using some sort of control mechanism. The cost savings do take time, there is the initial cost of deploying, adapting and hiring/training resources - but the savings exist.

What has been discussed in these texts are only two pragmatic facets of Open Source; there are more that have been missed (and may be addressed later) such as heritage, tradition, science, hobbyists, friendship, community and just plain fun - for the short-sighted pragmatic or Open Source afficiando alike these texts explain the methods and models but skip over the human factor . . .

. . . you need to figure that one out yourself.

1.2.0 Syntax and Context Grammar and Programming

Comparing a spoken language to most contemporary programming languages might be a reach. Is it possible that the structure of grammar might be comparable to programming? There are two extremely simple ways that grammar, English grammar in particular, can be compared to programming statements: syntax and context. Syntax

One definition of syntax is:
Syntax

That part of grammar which treats of the construction of sentences; the due arrangement of words in sentences in their necessary relations, according to established usage in any language. [1913 Webster] On the C Programming Language

The C language has very strict typing and checking. In a sense, the C language can be compared to when grammar is used in very strict terms or perhaps an easier way to think about it - not hackney. Not all languages, including English, apply the rule of being overly strict all of the time. In very simple sentences correct syntax is the rule of thumb.

```
....  
The boy is tall.  
....
```

Pretty straightforward, if the structure changes, and the rules for English broken, the sentence becomes very difficult to follow:

```
....  
Tall is the boy  
....
```

Is this saying the same thing? No, it might be asking a question or is just malformed by accident (or you are Yoda). Doing something similar in C can simply break it:

```
....  
char *boy = "tall";  
....
```

Works - but -

```
....  
"tall" = char *boy;  
....
```

does not. In general terms, strict grammatical and some programming syntax share a similar set of relations. A more complicated example might be looking at iteration. Iteration is not so easy to do in typed words because pages do not loop by themselves. Conversation can create an iterative function:

```
$ "Is it nice out today?"  
$ "no"  
$ "Then I am going to work"
```

Simple iteration; look for something and either act upon it, look for something else or go on.

```
/* in some other part of the program */
char *weather = "lousy";
if (strcmp(weather, "lousy"))
    work();
    play();
return 0;
```

Although the action of iteration will appear similar between many programming languages it is still important to note that strong typing and syntax are among the most important aspects of C. Understanding this makes looking at context all that much more interesting. Context

A good definition for context (within the context of this article) is: context

n 1: discourse that surrounds a language unit and helps to determine its interpretation [syn: {linguistic context}, {context of use}] 2: the set of facts or circumstances that surround a situation or event; "the historical context" [syn: {circumstance}] The Perl Programming Language

Whereas C is based on strict structure and typing, Perl is much more context based. By this, it means a basic Perl data type, the scalar, can be anything based on the context it is represented in. The similarity to grammar is the concept of context itself.

English context for some can be difficult to master because different words have different meanings in different contexts. Such as:

"To say I like that job is akin to loving glass eating." "To say I like that beer is similar to asking me if I mind eating to live."

The first sentence is a negative statement. The second statement positive (if not a lame attempt at humor). Both use the word like. The sentence connotation is distinguished by information that comes later and puts the word like in the proper context. By means of pure context the most basic Perl variable, a scalar, is similar:

```
$like = "yes";
$like = 1;
$like = 'no';
$like = `NO`;
```

All of those are a scalar of the same name but defined later in the statement by more information, in this case, the actual data itself surrounded by (or not) ticks, quotes or double quotes. What can be even more interesting (or distressing depending upon your point of view) is how Perl can change the literal context:

```
[jrf@vela:~$] cat foo.pl
$bar = "ooga wooga";
@foo = $bar;
print "@foo\n";
push (@foo, "booga googa");
push (@foo, 42);
print "@foo\n";
[jrf@vela:~$] perl foo.pl
ooga wooga
ooga wooga booga googa 42
```

The above is perfectly legal and of course quite useful.

Context also dictates that the actions of something later can affect the context of a word. In Perl, this can very easily be summarized:

```
sub somesub {
    if ("somedir") {
        return "somedir";
    } else {
        return 1;
    }
}
```

If the function is assigning a variable then that variable can have a totally different type of value depending on the circumstances. In truth, they are just bits and bytes of course and Perl magically translates the difference, semantically, however, they are two different types of data. Other Comparisons

Other programming languages can be laid side by side to metaphors. Object oriented programming is probably one of the most obvious programming language types that does so in a very literal sense. List Processing languages are based on recursive thought mechanisms (a simplification of course). Summary

To know something such as the difference between strict syntax and loose context can help a programmer to better understand why things work (or don't) the way they do.

2.0.0 The Bash Shell

The BASH shell or bourne again shell is the GNU version of the stock bourne shell (or simply sh) with GNU extensions. To date bash is the most popular Unix shell.

Bash is being used as the starting point in the examples as it has a low entry point and at the same time is pivotal to the system. What many Unix users are not aware of (but administrators, developers and home users are all too well aware) is the system shell is used to bring up and manage most services on a Unix system. This makes looking at shell scripting examples a little more interesting because knowledge here can be leveraged elsewhere.

2.0.1 Relatively Small Bash Scripts

2.0.1.i. Qtop

A shell script can do an amazing amount of work with a relatively small amount of typing. In the following script from Shelldorado (<http://shelldorado.com/>) gets a list of the top 15 processes and refreshes it every 5 seconds:

```
#!/bin/bash
DISPPROC=15
DELAY=5
clear
while (true)
do
    clear
    echo "-----"
    echo "                                Top Processes"
    uname -a
    uptime
    date
    echo "-----"
    /bin/ps aux | head -${DISPPROC}
    sleep $DELAY
done
```

Amazingly small yet it creates a nice display:

```
Top Processes Linux adm102 2.6.16.46-0.12-default #1 Thu May 17 14:00:09 UTC 2007 x86_64 x86_64
x86_64 GNU/Linux 1:08pm up 61 days 5:32, 4 users, load average: 0.49, 0.76, 0.70 Wed Nov 26 13:08:51
EST 2008
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.0	784	76	?	S	Sep26	11:06	init [3]
root	2	0.0	0.0	0	0	?	SN	Sep26	0:00	[ksoftirqd/0]
root	3	0.0	0.0	0	0	?	S<	Sep26	0:04	[events/0]
root	4	0.0	0.0	0	0	?	S<	Sep26	0:00	[khelper]
root	5	0.0	0.0	0	0	?	S<	Sep26	0:00	[kthread]
root	7	0.0	0.0	0	0	?	S<	Sep26	0:02	[kblockd/0]
root	8	0.0	0.0	0	0	?	S<	Sep26	0:00	[kacpid]
root	91	0.0	0.0	0	0	?	S	Sep26	11:28	[pdflush]
root	94	0.0	0.0	0	0	?	S<	Sep26	0:00	[aio/0]
root	93	0.0	0.0	0	0	?	S	Sep26	8:33	[kswapd0]
root	302	0.0	0.0	0	0	?	S<	Sep26	0:00	[cqueue/0]
root	303	0.0	0.0	0	0	?	S<	Sep26	0:00	[kseriod]
root	334	0.0	0.0	0	0	?	S<	Sep26	0:00	[kpsmoused]
root	674	0.0	0.0	0	0	?	S<	Sep26	0:00	[scsi_eh_0]

2.0.1.ii MD5 Script

The next script generates and md5 checksum of a file specified on the command line using the md5 binary. First a look at the surrounding pieces of the script; comments are where they are needed to explain what these do. Also note that functions are being used for the first time.

```
progname=${0##*/} # Capture the name of the script
toppid=$$ # Capture the PID
```

```

trap "exit 1" 1 2 3 15 # Trap on these exits

# We use this routine to bail out. Print our error then
# kill ourselves with great discrimination.
bomb()
{
    cat >&2 <<ERRORMESSAGE

    ERROR: @$
    *** ${progname} aborted! ***
    ERRORMESSAGE
    kill ${toppid}
    exit 1
}

```

Pretty simple so far; now the next two functions; generate the hash and a usage printer:

Based on the number being passed either generate a quiet sum or loud one

```

DoGen()
{
    if [ $3 -ne "0" ]; then
        md5 $1 > $2 || bomb "Could not gen md5 for ${1}"
    else
        md5 $1 | awk '{print $4}' > $2 || bomb "Could not gen md5 for ${1}"
    fi
}
# Print a usage message
Usage()
{
    cat <<__usage__
Usage: ${progname} [-f file-to-use] [-F]
Arguments:
    -f The file to get a signature from
    -o Output file (md5 is the default
Flags:"
    -F Create a signature using all of md5 output
__usage__
}

```

Note the usage message uses an atomic echo; that is all at once (which is why it is not indented). There is a logical reason for this; each echo, print, printf etc. requires a separate system call. One atomic print uses one call. While not a big deal now overtime such practice can add up to time saved.

Now on to the main part of the script. Here the options are cased in and the input file assigned, output of the sum, and what type:

```

FULLMD5FILE=0
OUTFILE="md5"
while [ "$#" -gt "0" ]
do
    opt="${1//-/}"; pt=`cat $opt | cut -c 1`
    case $1 in
        F) FULLMD5FILE=1 ;;
        f)      shift ; SRCFILE=$1 ;;
        o) shift ; OUTFILE=$1 ;;
        u) Usage ; exit 0 ;;
        *) echo "Syntax Error" ; Usage ; exit 1 ;;
    esac
done

```

```

shift
done

DoGen $SRCFILE $OUTFILE $FULLMD5FILE

exit 0

```

One little oddity about shell scripting: when using functions note that the function_name(parameter list...) nomenclature is not used. Programmers coming from other languages might find this a little hard to get used to at first. Now the whole listing:

```

programe=${0##*/} # Capture the name of the script
toppid=$$ # Capture the PID
trap "exit 1" 1 2 3 15 # Trap on these exits

# We use this routine to bail out. Print our error then
# kill ourselves with great discrimination.
bomb()
{
    cat >&2 <<ERRORMESSAGE
    ERROR: @$
    *** ${programe} aborted! ***
    ERRORMESSAGE
    kill ${toppid}
    exit 1
}

DoGen()
{
    if [ $3 -ne "0" ]; then
        md5 $1 > $2 || bomb "Could not gen md5 for ${1}"
    else
        md5 $1 | awk '{print $4}' > $2 || bomb "Could not gen md5 for ${1}"
    fi
}

Usage()
{
    cat <<__usage__
Usage: ${programe} [-f file-to-use] [-F]
Arguments:
    -f The file to get a signature from
    -o Output file (md5 is the default)
Flags:
    -F Create a signature using all of md5 output
__usage__
}

FULLMD5FILE=0
OUTFILE="md5"
while [ "$#" -gt "0" ]
do
opt="${1//-}"; pt=`cat $opt | cut -c 1`
case $1 in
    F) FULLMD5FILE=1 ;;
    f) shift ; SRCFILE=$1 ;;
    o) shift ; OUTFILE=$1 ;;
    u) Usage ; exit 0 ;;
    *) echo "Syntax Error" ; Usage ; exit 1 ;;
esac
shift

```

```
done  
DoGen $SRCFILE $OUTFILE $FULLMD5FILE  
exit 0
```

2.0.2 Bash and I/O

One of the jobs of the Unix shell is navigate and work with Unix filesystems. Naturally if one of the jobs of the shell is to work with filesystems and files then shell scripting is an ideal solution to the same end. The next two scripts tackle two distinctly different topics:

1. Finding header files without taxing a system.
2. Checking Network Filesystem(s)

2.0.2.i. Finding Header Files

Once upon a time there as a sysadmin who wanted to be able to quickly look for header files without using the find command across too many filesystems and so the hfind script was born. Outside of the core function the hfind script comes packed with the usual pieces needed with 2 extra globals, where to look and the maximum number of passes, that is when to stop looking across the \$ldpaths string:

```
program=${0##*/}
toppid=$$
ldpaths="/usr/include /usr/local/include /opt /usr"
passmax=4
trap "exit 1" 1 2 3 15

bomb()
{
    cat >&2 <<ERRORMESSAGE

    ERROR: @$
    *** ${program} aborted ***
    ERRORMESSAGE
        kill ${toppid}      # in case we were invoked from a subshell
        exit 1
}

usage()
{
    cat <<usage
Usage: ${program} [option arg][option]
Usage: ${program} [-f filename][-p passes] [-u]
Passes: Number of different paths to try and pass through.
usage
}

```

The next helper function simply prints out where the search is looking and some advice if someone needed to add (or subtract) the paths:

```
showpaths()
{
    echo "Current paths searched (in order):"
    for pth in ${ldpaths} ; do
        echo ${pth}
    done
    echo "Modify the \${ldpaths} variable at the top of the script if you like"
}

```

Now take a look at the main loop, not altogether different than previous examples but formatted to be a little more compact. This shows the flexibility of how shell scripts literally interpret what they are fed:

```
passes=1
verbose=0
while [ "$#" -gt "0" ]
do
    opt="{1//-}"
    opt=$(echo "${opt}" | cut -c 1 2>/dev/null)
    case $opt in
        f) shift;filename=$1;;
        p) shift;passes=$1;;
        s) showpaths;exit 0;;
        u) usage;exit 0;;
        v) verbose=1;;
        *) usage;exit1;;
    esac
    shift
done

if [ ! $filename ]; then
    echo "Error: No filename specified"
    usage
    exit 1
fi

search $passes $filename
```

Finally onto the core function of the script; the actual search which is relatively simple - it goes to the top of each path and looks only within that subdirectory:

```
search()
{
    pass=$1
    file=$2

    cnt=0

    for pth in ${ldpaths} ; do
        cnt=$((cnt+1))
        if [ "$verbose" -gt "0" ]; then
            echo "Searching for ${file} in ${pth}"
        fi

        find $pth -type f -name $file
        if [ "$cnt" -eq "$pass" ]; then
            break;
        elif [ "$cnt" -eq "$passmax" ]; then
            echo "Search exhausted"
            exit 0
        fi
    done

    return $?
}
```

It is worth noting at this point there should be a pattern that can be observed regarding how the hfind script was presented. Outside of the main portion of the script all other functions were considered helper functions. The core function is really the key to the entire script, and this is a central axiom of Unix programming that

much of a program exists outside of a core algorithm to either support it or in the case of the main function to get information needed by the core algorithm. This idea of do one thing well will continue to crop up throughout this book. Now for the finished program:

```

program=${0##*/}
toppid=$$
ldpaths="/usr/include /usr/local/include /opt /usr"
passmax=4

trap "exit 1" 1 2 3 15

#-----
# bomb - Simple death routine; display ERRORMESSAGE, kill toppid and exit.
#
# requires: ERRORMESSAGE
# returns : exit 1
#-----
bomb()
{
    cat >&2 <<ERRORMESSAGE

ERROR: @$
*** ${program} aborted ***
ERRORMESSAGE
    kill ${toppid}      # in case we were invoked from a subshell
    exit 1
}

#-----
# search - Search $ldpaths
#
# requires: Npass
# returns : $?
#-----
search()
{
    pass=$1
    file=$2

    cnt=0

    for pth in ${ldpaths} ; do
        cnt=$((cnt+1))
        if [ "$verbose" -gt "0" ]; then
            echo "Searching for ${file} in ${pth}"
        fi

        find $pth -type f -name $file
        if [ "$cnt" -eq "$pass" ]; then
            break;
        elif [ "$cnt" -eq "$passmax" ]; then
            echo "Search exhausted"
            exit 0
        fi
    done

    return $?
}

#-----

```

```

# usage - simple usage print
#-----
usage()
{
    cat <<usage
Usage: ${program} [option arg][option]
Usage: ${program} [-f filename][-p passes] [-u]
Passes: Number of different paths to try and pass through.
usage
}

#-----
# showpaths - print all of the currently searched paths
#-----
showpaths()
{
    echo "Current paths searched (in order):"
    for pth in ${ldpaths} ; do
        echo ${pth}
    done
    echo "Modify the \${ldpaths} variable at the top of the script if you like"
}

passes=1
verbose=0
while [ "$#" -gt "0" ]
do
    opt="${1//-}"
    opt=$(echo "${opt}" | cut -c 1 2>/dev/null)
    case $opt in
        f) shift;filename=$1;;
        p) shift;passes=$1;;
        s) showpaths;exit 0;;
        u) usage;exit 0;;
        v) verbose=1;;
        *) usage;exit 1;;
    esac
    shift
done

if [ ! $filename ]; then
    echo "Error: No filename specified"
    usage
    exit 1
fi

search $passes $filename

```

There are a lot more comments in the full script. Good commenting even if it is as rudimentary as simply explaining what a section of code is doing is as important for the author as it is someone who may wish to modify and use it.

2.0.2.ii Check NFS

The next example is not an entire program... yet - we will be revisiting it at the end of this section. Instead only some functions which use some tricks. We want to first check to see if the count of actual mounted filesystems matches against a known minimum by counting the number of nfs mounts that should be mounted versus the number of nfs mounts that are mounted. The trick? ... we need to check them on a remote host. There is an assumption: ssh public key exchange is working for the account being used - as with previous

examples detailed commenting is used to by pass lengthy discussion:

```
chknfs()
{
host=$1 # Pass in the name or IP of the host

# SSH to the remote host and grep out of /etc/fstab any line
# that is not a comment and is a nfs filesystem ONLY.
nfs_fstab=$(ssh $host grep "^[^#].*nfs" /etc/fstab |
awk '{print $2}' 2>/dev/null)

# Check the ACTUAL number of NFS mounts currently mounting using
# the mount command: DO NOT USE df - it likes to hangs
nfs_mount=$(ssh $host mount | grep nfs | awk '{print $3}' 2>/dev/null)

# Compare the mounts (1 for 1) to see if a NFS mount is mounted more
# than once or is not mounted at all
for i in $nfs_fstab; do # For every valid nfs mount in fstab
    matches=0 # no matches yet
    for j in $nfs_mount; do # Increment matches if we find it
        if [ $i == $j ]; then
            matches=$((matches+1))
        fi
    done

# Print out an alarm if needed based on the number of
# matches found (or not as the case may be)
# Ruh-roh - no matches - we are missing one
if [ "$matches" -eq 0 ]; then
    cerror $host "nfs: ${i} not mounted"
    return 1
# Crap; must be an old version of nfs... too many mounts
elif [ "$matches" -gt 1 ]; then
    cerror $host "nfs: ${i} is mounted multiple times"
    return 2
fi
done

return 0
}
```

The observant will note there is a lot that can be done with the above function - for instance what if the settings need to be checked? What if the actual version needs to be checked? All in all however, it is easy see how powerful shell programming can be especially for diagnostics.

2.0.3 Bash Network and OS Scripts

Even though the nfs check uses SSH, a better script that deals with networking would be nice plus an example of how shell scripts interact closely with Operating System services are the next and final scripts before the program.

2.0.3.i Remote Sync Script

The rsync utility can be used to synchronize a source and destination directory either on the same machine or across the network using some sort of protocol mechanism. For our example the secure shell protocol is used again - the assumption for keys is not mandatory - if passwords are permitted the script will request the user enter a password as soon as the synchronization begins. Unlike previous scripts the main concern is providing as much information as possible to rsync in order to simplify an rsync operation of this type. Additionally the POSIX getopt function is used which means there will be an extra safety check. Here is the top part of the script:

```
#!/bin/sh
progrname=${0##*/}
toppid=$$
PROTO=ssh
UTIL=rsync
UTIL_FLAGS="-az --delete -e $PROTO"
bomb()
{
    cat >&2 <<ERRORMESSAGE

    ERROR: @$
    *** ${progrname} aborted ***
    ERRORMESSAGE
        kill ${toppid}
        exit 1
}
# We check to see if all the needed binaries are in PATH
for i in $PROTO $UTIL
do
    if ! type ${i} >/dev/null; then
        bomb "${i} not found"
    fi
done
# make sure our shell supports getopt
if ! type getopt >/dev/null 2>&1; then
    bomb "/bin/sh shell is too old; try ksh or bash"
fi
```

Since there are some non-standard tools being used some more safety checks are piled on. Once all of that is done; it is a matter of building up the string needed by parsing input and passing along all of the information to the command:

```
while getopt s:d: ch; do
    case ${ch} in
        s) SRC=${OPTARG};;
        d) DST=${OPTARG};;
    esac
done
shift ${(${OPTARG} - 1)}
```

```

$UTIL $UTIL_FLAGS $SRC $DST ||
    bomb "Could not run ${UTIL}  ${UTIL_FLAGS} ${SRC} ${DST}"

exit 0

```

Simple and sweet and ironically, all it is doing is the following:

```

#
# rsync -az --delete -e ssh hostname_or_ip:/path/to/src /path/to/dst
#

```

And reducing that to:

```

#
# script_name host:/path /path/to/dst
#

```

Which generally is a lot easier to read in the crontab.

2.0.3.ii Service Script

Several Linux distributions ship with a neat utility called the service utility. All this utility does is locate and execute a init script and any arguments. Surprisingly I have found this utility to be very handy and not on BSD or several other Linux distributions so of course - I wrote my own based on a version from RedHat.

In order to accomodate multiple platforms the script looks for the init directory; there should be only one:

```

for dir in /etc/rc.d /sbin/init.d /etc/init.d ; do # set init dir
    if [ -d "${dir}" ];then
        SERVICEDIR=${dir}
    else
        echo "No init script directory found" && exit 1
    fi
done

```

Next check to make sure there are arguments:

```

if [ $# -eq 0 ]; then
    echo "${USAGE}" >&2
    exit 1
fi

```

Go somewhere safe then parse out the arguments. If a valid operation was passed attempt to perform the operation, otherwise parse other arguments or error out:

```

cd / # go someplace safe
while [ $# -gt 0 ]; do
    case "${1}" in
        --help | -h | --h* ) # Need help?
            echo "${USAGE}" >&2
            exit 0
            ;;
        --list | -l | --l* ) # Lets see what is in the init dir
            cd $SERVICEDIR && ls
            exit 0
            ;;
    esac
done

```

```

--version | -V ) # What version is this?
    echo "${VERSION}" >&2
    exit 0
                ;;
*)
    SERVICE="${1}" # Try to perform the service op
    COMMAND="${2}"
    if [ -x "${SERVICEDIR}/${SERVICE}" ]; then
        env -i LANG=$LANG PATH=$PATH TERM=$TERM \
            "${SERVICEDIR}/${SERVICE}" ${COMMAND} || exit 1

    else

        echo "${SERVICE}: unrecognized service" >&2
        exit 1

    fi
                ;;
esac
done

exit 2

```

Following is the full listing plus the missing information from the top:

```

#!/bin/sh
# Script -----
# service - A slightly enhanced version of the redhat service script.
#           Supports several different service locations.
#
#-----
PATH="/sbin:/usr/sbin:/bin:/usr/bin:/usr/X11R6/bin" ; export PATH
VERSION="`basename $0` ver. 0.91"
USAGE="Usage: `basename $0` < option > | [ service_name command ]"
SERVICE=
COMMAND=

for dir in /etc/rc.d /sbin/init.d /etc/init.d ; do # set init dir
    if [ -d "${dir}" ];then
        SERVICEDIR=${dir}
    else
        echo "No init script directory found" && exit 1
    fi
done

if [ $# -eq 0 ]; then
    echo "${USAGE}" >&2
    exit 1
fi

cd /
while [ $# -gt 0 ]; do
    case "${1}" in
        --help | -h | --h* )
            echo "${USAGE}" >&2
            exit 0
            ;;
        --list | -l | --l* )
            cd $SERVICEDIR && ls
            exit 0
            ;;
        --version | -V )
            echo "${VERSION}" >&2
            exit 0
            ;;
    esac
done

```

```

*)
SERVICE="${1}"
COMMAND="${2}"
if [ -x "${SERVICEDIR}/${SERVICE}" ]; then
    env -i LANG=$LANG PATH=$PATH TERM=$TERM \
        "${SERVICEDIR}/${SERVICE}" ${COMMAND} || exit 1
else
    echo "${SERVICE}: unrecognized service" >&2
    exit 1
fi
;;
esac
done
exit 2

```

Not too difficult and easily ported from one platform to the next.

2.0.4 Bash Program: cnchk

Expanding on the example from the Filesystem and I/O section, the program in this example runs a battery of tests on another Linux host (or hosts). Noting the central algorithm axiom, in the cnchk (or compute node check) the central algorithm is the ability to call the battery tests easily and add to them with relative ease.

To keep the size of the program to a minimum we are performing three diagnostic tests on each system:

1. The nfs check
2. Make sure root is readable
3. Check to see if any local filesystems are at 100 percent

The nfs check is already done, so here are the other two checks heavily commented:

```
chkdsk() # The name of the function
{
  host=$1 # Checking this host

  # Get fs usage in percentages from local filesystems ONLY
  # (if there are nfs problems they should not interfere
  fsperc=$(ssh $host df -l | grep -v Use | awk '{ print $5 }' 2>/dev/null)

  # Check each local filesystem percentage to see if it is 100%
  for i in $fsperc; do
    if [ ${i} = "100%" ]; then
      echo $i
      echo -n "filesystem: a local filesystem is full"
      return 1 # We hit a snag - let the caller know
    fi
  done

  return 0 # We are okay
}

chkwrite()
{
  host=$1 # Checking this host

  ssh $host touch /tmp/nodechk.$$ # See if we can create a file

  # If we couldn't create a file throw an alarm
  if [ $? -gt 0 ]; then
    echo "${host} filesystem: Read only root filesystem"
    return 1 # Return bad news
  else
    ssh $host rm /tmp/nodechk.$$ # Otherwise clean up
    fi

    return 0 # we are okay
}
```

A few things worth noting. The names of the check functions are consistent (albeit lame) - this is on purpose. In the larger listing it helps to differentiate helper functions from the test cases. Now armed with the common bomb routine all this program needs is the parser. To mix things up lets create the the program in such a way that it can do either one host, a range of hosts with a common name or a comma delimited list of hosts:

Input parsing - the usage explains what each one does

```
OPER=""
if [ $# -gt 0 -a "$1" = "-h" ];then
usage
exit 0
fi
if [ $# -gt 0 -a "$1" = "--help" ];then
usage
exit 0
fi

while [ "$#" -gt "0" ]; do
case $1 in
# Create a list of hosts as n1,n2,n3
--node=*)
    NODELIST="${1#*=}"
    NODELIST="${NODELIST//,/ }"
    OPER="list"
    ;;
--node|-n)
    NODELIST="$2"
    shift
    NODELIST="${NODELIST//,/ }"
    OPER="list"
    ;;
# Create a range of hosts as hostname[n-N]
--range=*)
    RANGE="${1#*=}"
    RANGE="${RANGE//-/ }"
    RANGE="${RANGE//[a-z]/ }"
    PREFIX="${1#*=}"
    PREFIX="${PREFIX%%[0-9]*}"
    OPER="range"
    ;;
--range|-r)
    RANGE="$2"
    RANGE="${RANGE//-/ }"
    RANGE="${RANGE//[a-z]/ }"
    PREFIX="${2#*=}"
    PREFIX="${PREFIX%%[0-9]*}"
    OPER="range"
    shift
    ;;
# Default - just enter a hostname
*)
    NODELIST="$1"
    NODELIST="${NODELIST//,/ }"
    OPER="list"
    ;;
esac
shift
done
```

The program just got a lot more complex. There are essentially two operating modes- a list of nodes or a range of nodes which have to be dealt with differently. Essentially the nodelist can be iterated over while the range just counted; how to differentiate? - that is what the OPER variable was for:

```
# If a nodelist was specified, just iterate through it
if [ ${OPER} = "list" ]; then
for node in $NODELIST;do
```

```

        for diag in chknfs chkwrite chkdisk ; do
            $diag $node
            [ $? -gt 0 ] && continue    # Node fails: report and skip remaining
        done
    done
elif [ ${OPER} = "range" ]; then # Range specified
cur=$(echo ${RANGE} | awk '{ print $1 }' 2>/dev/null)    # Start node
end=$(echo ${RANGE} | awk '{ print $2 }' 2>/dev/null)    # Last node
prefix=$(echo ${PREFIX} | awk '{ print $1 }' 2>/dev/null) # Node prefix

while [ "$cur" -le "$end" ]; do # For the duration of the range

    for diag in chknfs chkwrite chkdisk ; do
        $diag $node
        if [ $? -gt 0 ]; then    # If we fail a test take
            cur=$((cur+1)) # actions based on flags
            continue          # and skip remaining
        fi
    done
done
else
    echo "Error! Nodes improperly specified"
    exit 2
fi
exit 0

```

Not so bad after all - notice anything missing? The bomb routine is not there. We do not want to completely bomb out if one node or check fails - we just want to know about it and move on. Also note some redundant typing; we could group the checks into one variable to save time later; for instance:

```

...
CHECKS="chknfs chkdisk chkwrite"
...

```

Then replace them in the for loops where they are explicitly typed in. Now for the final full listing:

```

#!/bin/bash
# Script -----
# Program      : cncheck (Compute Node CHECK)
# Author       : Jay Fink <fink.jr.1@pg.com>
# Purpose      : Run a battery or selective tests on node(s).
#-----
PROG=${0##*/}
TOPPID=$$
HOST=$(hostname 2>/dev/null)
trap "exit 1" 1 2 3 15

#-----
# chkdisk - Check local filesystem, if over the threshold send a message.
#           SSH to the host, parse df -lhP output for 100%.
#
# requires: hostname
# returns:  A 1 if a fs is > 100% else a 0
#-----
chkdisk() # The name of the function
{
host=$1 # Checking this host

# Get fs usage in percentages from local filesystems ONLY
# (if there are nfs problems they should not interfere

```

```

fsperc=$(ssh $host df -l | grep -v Use | awk '{ print $5 }' 2>/dev/null)

# Check each local filesystem percentage to see if it is 100%
for i in $fsperc; do
    if [ ${i} = "100%" ]; then
        echo $i
        echo -n "filesystem: a local filesystem is full"
        return 1 # We hit a snag - let the caller know
    fi
done

return 0 # We are okay
}

#-----
# chkwrite - Make sure / is read/writable.
#           SSH onto the host, create a tmp file then delete it.
#
# requires: hostname
# returns:  1 if the filesystem is r-o else a 0
#-----
chkwrite()
{
host=$1 # Checking this host

ssh $host touch /tmp/nodechk.$$ # See if we can create a file

# If we couldn't create a file throw an alarm
if [ $? -gt 0 ]; then
    echo "${host} filesystem: Read only root filesystem"
    return 1 # Return bad news
else
    ssh $host rm /tmp/nodechk.$$ # Otherwise clean up
    fi

    return 0 # we are okay
}

#-----
# chknfs - First do a nfs numeric count, then ensure what is listed in fstab
#          is in fact mounted.
#
# requires: hostname
# returns:  1 if a problem was encountered else a 0
#-----
chknfs()
{
host=$1

nfs_fstab=$(ssh $host grep "^[^#].*nfs" /etc/fstab |
    awk '{print $2}' 2>/dev/null)

nfs_mount=$(ssh $host mount | grep nfs | awk '{print $3}' 2>/dev/null)

for i in $nfs_fstab; do

    matches=0
        for j in $nfs_mount; do
            if [ $i == $j ]; then
                matches=$((matches+1))
            fi
        done
done

```

```

        if [ "$matches" -eq 0 ]; then
            cerror $host "nfs: ${i} not mounted"
            return 1
        elif [ "$matches" -gt 1 ]; then
            cerror $host "nfs: ${i} is mounted multiple times"
            return 2
        fi
    done

return 0
}

#-----
# usage - Usage message
#-----
usage()
{
if [ -n "$*" ]; then
    echo " "
    echo "${PROG}: $*"
fi
    cat <<usage
${PROG} [option argument][option]
${PROG} [nodename] [option arg]
${PROG} [--node=NODE1,NODE2|-node NODE1,NODE2|-n NODE1,NODE2]
        [--range=NODE1-NODEn|--range NODE1-NODEn|-r NODE1-NODEn]
Examples:
    Check nodes 2-15 on dev cluster, run in verbose and log.
${PROG} --range=dev1-16
${PROG} -r dev1-16
usage
}

#-----
# Main Loop
#-----
# Input parsing - the usage explains what each one does
OPER=""
if [ $# -gt 0 -a "$1" = "-h" ];then
usage
exit 0
fi
if [ $# -gt 0 -a "$1" = "--help" ];then
usage
exit 0
fi

while [ "$#" -gt "0" ]; do
case $1 in
# Create a list of hosts as n1,n2,n3
--node=*)
        NODELIST="${1#*=}"
        NODELIST="${NODELIST//,/ }"
        OPER="list"
        ;;
--node|-n)
        NODELIST="$2"
        shift
        NODELIST="${NODELIST//,/ }"
        OPER="list"
        ;;

```

```

# Create a range of hosts as hostname[n-N]
--range=*)
    RANGE="{1#*}"
    RANGE="{RANGE//-/}"
    RANGE="{RANGE//[a-z]/}"
    PREFIX="{1#*}"
    PREFIX="{PREFIX%%[0-9]*}"
    OPER="range"
    ;;
--range|-r)
    RANGE="$2"
    RANGE="{RANGE//-/}"
    RANGE="{RANGE//[a-z]/}"
    PREFIX="{2#*}"
    PREFIX="{PREFIX%%[0-9]*}"
    OPER="range"
    shift
    ;;
# Default - just enter a hostname
*)
    NODELIST="{1}"
    NODELIST="{NODELIST//,/}"
    OPER="list"
    ;;
esac
shift
done

# If a nodelist was specified, just iterate through it
if [ ${OPER} = "list" ]; then
for node in $NODELIST;do
    for diag in chknfs chkwrite chkdisk ; do
        $diag $node
        [ $? -gt 0 ] && continue # Node fails: report and skip remaini
ng tests
    done
done
elif [ ${OPER} = "range" ]; then # Range specified
cur=$(echo ${RANGE} | awk '{ print $1 }' 2>/dev/null) # Start node
end=$(echo ${RANGE} | awk '{ print $2 }' 2>/dev/null) # Last node
prefix=$(echo ${PREFIX} | awk '{ print $1 }' 2>/dev/null) # Node prefix

while [ "$cur" -le "$end" ]; do # For the duration of the range

    for diag in chknfs chkwrite chkdisk ; do
        $diag $node
        if [ $? -gt 0 ]; then # If we fail a test take
            cur=$((cur+1)) # actions based on flags
            continue # and skip remaining
        fi
    done
done
else
    echo "Error! Nodes improperly specified"
    exit 2
fi
exit 0

```

2.0.5 Bash Program: vmware init

The free vmware product, vmware-server (formerly GSX) does not have auto power on for certain guests. A simple workaround for not being able to auto power on guests using the vmware interface is call the vmware command line utility at boot up using the local init function. A better way is to write an init script to handle the start up and possibly other functions. This text will examine a simple method to create a control script for managing power functions in vmware-server. The Ultra Cheap Method

The easiest method without bothering to write a ctl script or wrapper is to find the vmid of the vmware guest to start up and embed a direct call using the vmware-vim-cmd interface in the systems local init script (generally /etc/rc.local). First get the id of the guest:

```
# vmware-vim-cmd vmsvc/getallvms
Vmid      Name                                     File
112      freebsd7                                [standard] freebsd7/freebsd7.vmx \
        freebsd64Guest    vmx-07    vela: irc server running freebsd7
208      netbsd5.99.10_amd64 [standard] \
        netbsd5.99.10_amd64/netbsd5.99.10_amd64.vmx\
        otherGuest64      vmx-07
240      opensuse11-prime      [standard] \
        opensuse11-prime/opensuse11-prime.vmx \
        suse64Guest        vmx-07
96       freebsd8                                [standard] \
        freebsd8/freebsd8.vmx      freebsd64Guest \
        vmx-07      pyxis: freebsd-8.0 development server
```

A bit messy looking but the ids are easy enough, now just add it to the local init script:

```
# we get the VMID using vmware-vim-cmd vmsvc/getallvms
if [ -x /usr/bin/vmware-vim-cmd ]; then
    echo "Starting Guest VMID 112, sleeping for 16 seconds"
    sleep 16
    /usr/bin/vmware-vim-cmd vmsvc/power.on 112
fi
```

Of course that is far too simple, one of the most common operations performed on vmware guests is powering on, powering off and resetting the host. A good sysadmin is lazy, so it is time to draft an ipmitool-like ctl script for controlling the power of the guests.

Config File

Because the format of vmware-vim-cmd vmsvc/getallvms does not show hostname or IP address mapping the vmid to host or IP address saves some time. Following is the format of our gsxhosts file:

```
# hostname VMID
vela      112
carina    208
pyxis     96
```

The format may not make sense now, however, when it is parsed in the script the overly simple format will make more sense. To get things started setup the hosts file, the vmsvc command, the usage message and a small error exit routine to make error handling simple - note the script supports all power operations:

```
#!/bin/sh
```

```

# gsx-ipmi - An ipmilike shell wrapper for vmware-vim-cmd vmsvc/power.*
HOSTSFILE=/etc/gsxhosts # This can be anywhere the admin likes
VIMVC=" vmware-vim-cmd vmsvc/" # We just tack on the oper
usage()
{
    if [ -n "$*" ]; then
        echo " "
        echo "${PROG}: $*"
    fi
    cat <<usage
${PROG} [host][oper cmd][[-u]
${PROG} [host][power getstate|hibernate|off|on|reboot|shutdown|suspend][[-u]
Commands:
  getstate      Display the current power state of the guest
  hibernate     Place the guest power into hibernate mode (OS must support)
  off           Power off the guest
  on            Power on and boot up the guest
  reboot        Normal reboot of the guest
  reset         Power reset (cold) the guest
  shutdown     Normal shutdown of the guest
  suspend       Place the guest into suspended mode
Notes:
  The user must have appropriate privileges to perform power operations on
  guests.
  usage
}
# Only call this if there was an input error because it displays the usage
# message
error_exit()
{
    message=$1
    exit_code=$2

    echo $message
    usage
    exit $exit_code
}

```

Take note of the usage, the script must specify an operation - the reason for this is to be able to add functionality later on. Even though for now the scope of the script is limited to power commands, done properly, the script could later have other vimvc operations added to it and the usage is similar to the ipmi command. Now onto the meat of the script, believe it or not it is straightforward, since all that has to be done is to tack on to the command string "power.\$operation" there are 3 basic steps:

1. validate input
2. ascertain the vmid
3. attempt to execute the command

3.

First the validation:

```

# Input parsing - the usage explains what each one does
if [ $# -gt 0 -a "$1" = "-u" ];then
    usage
    exit 0
fi

guest=$1

```

```

oper=$2
subcmd=$3

if [ ! $guest ]; then
    echo "Error: No guest specified"
    usage
    exit 1
fi

[ ! $guest ] << error_exit "No guest specified" 1
[ ! $oper ] << error_exit "No operation specified" 1
[ ! $subcmd ] << error_exit "No subcommand specified" 1

```

not too difficult, next try to get the vmid using grep and awk, this is where the simple file format comes into play:

```

#
vmid=`grep $guest $HOSTSFILE|awk '{print $2}'`
[ ! $vmid ] << error_exit "${guest} did not match anything in $HOSTSFILE" 2

```

With the vmid in hand the last step is to determine the operation then execute:

```

#
case $oper in
    power)
        $VIMVC"power.$subcmd $vmid 2>/dev/null
        if [ $? -gt 0 ]; then
            error_exit "$subcmd failed" 1
        fi
        ;;
    *)
        error_exit "Invalid operation" 2
        ;;
esac
exit 0

```

The power case could be more exotic but to keep from having to include all of the valid commands (even as a sed compare) we just fail. Note how by casing in the operation the doorway is left open to add other vimvc commands.

The Full Script

```

#!/bin/sh
# gsx-ipmi - An ipmilike shell wrapper for vmware-vim-cmd vmsvc/power.*
HOSTSFILE=/etc/gsxhosts # This can be anywhere the admin likes
VIMVC=" vmware-vim-cmd vmsvc/" # We just tack on the oper

usage()
{
    if [ -n "$*" ]; then
        echo " "
        echo "${PROG}: $*"
    fi
    cat <<usage
${PROG} [host][oper cmd][[-u]
${PROG} [host][power getstate|hibernate|off|on|reboot|shutdown|suspend][[-u]
Commands:
    getstate      Display the current power state of the guest

```

```

hibernate    Place the guest power into hibernate mode (OS must support)
off          Power off the guest
on           Power on and boot up the guest
reboot      Normal reboot of the guest
reset       Power reset (cold) the guest
shutdown    Normal shutdown of the guest
suspend     Place the guest into suspended mode

```

Notes:

The user must have appropriate privileges to perform power operations on guests.

usage

```

}

# Only call this if there was an input error because it displays the usage
# message
error_exit()
{
    message=$1
    exit_code=$2

    echo $message
    usage
    exit $exit_code
}

# Input parsing - the usage explains what each one does
if [ $# -gt 0 -a "$1" = "-u" ];then
    usage
    exit 0
fi

guest=$1
oper=$2
subcmd=$3

if [ ! $guest ]; then
    echo "Error: No guest specified"
    usage
    exit 1
fi

[ ! $guest ] << error_exit "No guest specified" 1
[ ! $oper ] << error_exit "No operation specified" 1
[ ! $subcmd ] << error_exit "No subcommand specified" 1

vmid=`grep $guest $HOSTSFILE|awk '{print $2}'`
[ ! $vmid ] << error_exit "${guest} did not match anything in $HOSTSFILE" 2

case $oper in
    power)
        $VIMVC"power."$subcmd $vmid 2>/dev/null
        if [ $? -gt 0 ]; then
            error_exit "$subcmd failed" 1
        fi
        ;;
    *)
        error_exit "Invalid operation" 2
        ;;
esac

exit 0

```

The end

2.1.0 The Perl Programming Language

The Practical Extraction and Reporting Language was invented by Larry Wall as a means to provide more solid programming constructs combined with the ease of scripting in making administration and system programming easier. One way to look at Perl is a middle ground between C and shell scripting which makes it an ideal stepping stone between shell scripting and C programming.

In this section a similar format to shell scripting and programming is used: several smaller examples separated by type with a final full fledged program.

2.1.1 Small Perl Script Examples

The first two examples are very small:

The first example comes from an old Matt's Script Archive example (<http://www.scriptarchive.com/>) while the second script is more of a function but the examination is using it as a program unto itself.

2.1.1.i Nuke M's

One of the big problems of old (not so much these days) was when users would transfer a text file from a FAT32 or WINNT filesystem over to a Unix system the file would retain the non-Unix end of line control characters; CTRL+Ms. The following snippet alleviates the problem, just run the script on the Unix host:

```
#!/usr/bin/perl
while (<>) {
    $_ =~ s/\cM\n/\n/g;
    print $_;
}
```

The program simply prints out the lines changed; but as is plain to see it opens the whole file and using a reassignment through regular expression strips off the control M and replaces it with a Unix new line. A simple yet powerful example of Perl.

2.1.1.ii Return the Contents of File

Perl is great at opening, reading and arranging the contents of a file internally or directories for that matter. In the next segment of code a file is read into an array, sorted, then printed out:

```
open FILE "/path/to/file" || die "Cannot Open File"
my @file_contents = <FILE>;
close FILE;
sort (@file_contents);
foreach(@file_contents) {
    print $_;
}
```

The first line attempts to open a file handle (a pointer to a file) called FILE using the path to a file or die and print a nice message. Next assign the file contents to an array, close the open file handle, sort and finally print it all out in a foreach loop.


```

    @flist = <FILE>;
    close FILE;

    return(@flist); # send the list back to the caller
}

```

2.1.2.ii Logging Stop and Restart Functions

In any program it is often useful to have file logging for a variety of purposes. The next two small functions start and restart logging respectively; the operations should look familiar (file opening):

```

sub start_logging{
    local($log_file) = @f[0];
    open(LOG,">$logfile");
    if(LOG){
        $log=1;
        return 1;
        $|=1;
    }
}

sub restart_logging{
    local($log_file) = @f[0];
    open(LOG,">>$logfile");
    if(LOG){
        $log=1;
        return 1;
    }
}

```

In perl functions are called subroutines - hence the sub before the function name. Also, to guarantee scope two pre-declarations can be used, local keeps a variable local to the file and my keeps it local to the context.

2.1.3 Network and OS

As illustrated earlier, Perl has access to many network and OS capabilities. Even with access to system interfaces Perl can be particularly useful on systems that allow easy access to system information such as Linux kernel based distributions. In addition to working with OS details Perl itself can be used as a daemon (or service in non Unix parlance).

2.1.3.i IP Connection Tracker

Linux systems have a firewall package called iptables. One problem that used to occur on Linux kernel based systems was the IP connection tracker in the kernel would become overloaded and start dropping valid packets. The following script does several things:

- Checks to see if the connections are getting too close to the maximum.
- If they are getting too close make sure there is sufficient memory to expand the connection tracker bucket.
- Add space to the connection tracker bucket.

Note that for brevity the `load_file()` sub routine discussed earlier is not repeated:

```
# Check the current connections situation...
sub check_ip_contrack {
    my $ip_contrack_max = `cat /proc/sys/net/ipv4/ip_contrack_max`;
    my $ip_contrack_cur = `wc -l /proc/net/ip_contrack`;
    my $hi_water_mark = ($ip_contrack_max * .6);
    my $ip_contrack_hard_limit = get_mem_max();

    if ($ip_contrack_cur >= $hi_water_mark) {
        my $new_value = ($hi_water_mark * 2);

        if (($new_value * 65000) >= $ip_contrack_hard_limit) {
            print "Error! IP CONNTRACK HAS REACHED THE 70% of RAM HIMARK!\n";
            exit 0;
        } else {
            system("echo $new_value > /proc/sys/net/ipv4/ip_contrack_max");
        }
    }
}

# Calculate how much memory we can gobble up for contrack
sub get_mem_max {
    my @kmeminfo = load_file("/proc/meminfo");
    my $ram_total = @kmeminfo[3];

    $ram_total =~ s{MemTotal:}{};
    $ram_total =~ s{kB}{};
    $ram_total =~ s{MB}{};

    return (.7 * $ram_total)*.06;
}

check_ip_contrack(); # No input required - just run
```

The above script could be run periodically by hand or simply scheduled using the cron system scheduler.

2.1.3.ii Creating A Daemon Process With Perl

he hypothetical scenario is simple, check for a subdirectory under /mnt/net, specifically, ajaxx and then log the status. The Initial Prototype

To see if it is even feasible, a very simple program is banged out just to get the concept down:

```
#!/usr/bin/perl
my $DELAY = 300;
my $mount_point = "/mnt/net/ajaxx";
unless(fork()) {

    unless(fork()) {
        while(1) {
            if ( -d "$mount_point" ) {
                print "Mount point $mount_point is present\n";
            } else {
                print "Mount point $mount_point missing\n";
            }
            sleep $DELAY;
        }
    }
}
```

An experienced Perl programmer probably sees one glaring flaw with the code (even though it does work):

```
unless(fork()) {
    unless(fork()) {
```

Has two problems:

- It is not using setsid to get a new pid [1]
- The module POSIX with setsid gets rid of the need for a two unless statement where it detaches.

It works, however, so a good starting point has been established. Note the \$DELAY which is in seconds.
Version 0.2

Now it is time to start plugging in the desired parts:

```
use strict;
use POSIX qw(setsid);
```

Use strict checking and setsid from POSIX.

```
my $DELAY      = 300;
my $MNT        = "/mnt/net/ajaxx";
my $LOG        = "/tmp/mntchkd.log";
my $PROGRAM    = "mntchkd";
```

5 minute delay, the directory being checked on, logfile location and the name of the program.

```

sub appendfile {
    my ($fp, $msg) = @_;

    if (open(FILE, ">>$fp")) {
        print FILE ("$msg\n");
        close FILE;
    }
}

```

A very simple logging utility that appends to a logfile by opening, appending then closing. The actual filename is defined by the first argument and assigned as fp. The second argument is a full string with the error message. Then a open and append to the file.

```

sub insert0 {
    my ($date) = shift;

    if ($date < 10) {
        return "0$date";
    }

    return $date;
}

```

Insert zeros into a date part. Helps to make a consistent column format for dates in the logfile.

```

sub longfmt {
    my ($sec, $min, $hour, $mday, $mon, $year,
        $yday, $yday, $iddst) = localtime(time);
    my $datestring;

    $year += 1900;
    $mon++;
    $mon = insert0($mon);
    $mday = insert0($mday);
    $min = insert0($min);
    $datestring = "$year-$mon-$mday $hour:$min";

    return($datestring);
}

```

A function that sets up and formats a date string for a log entry. Note the entire date string format, it can be customized easily.

```

unless(my $pid = fork()) {
    exit if $pid;
    setsid;
    umask 0;
    my $date = longfmt();
    appendfile($LOG, "$date Starting $PROGRAM");
    while(1) {
        $date = longfmt();
        if ( -d "$MNT" ) {
            appendfile($LOG, "$date Mount point $MNT present");
        } else {
            appendfile($LOG, "$date Mount point $MNT missing");
        }
    }
    sleep $DELAY;
}
}

```

Now the fun part. A simple fork, then assign the pid using setsid. Next setup a umask of 0 and an initial date for the "start" log message. The message is sent and a forever while loop is started. Note that the date is formatted at each iteration of the loop.

Last and not least, actually check for the subdirectory and sleep for the predefined delay.

2.1.4 Perl Program: Host Watch Daemon

Expanding on previous examples in this section; the Perl program in this section daemonizes and checks for processes on another host (or to see if the host is alive).

At the beginning is our initialization with descriptions:

```
#!/usr/bin/perl
##
# Perl daemon process to check a heartbeat
##
use strict;
use POSIX qw(setsid);
my ($FALSE,$TRUE) = (0,1); # A boolean w/o declaration
my $DELAY          = 30; # The delay in seconds between checks
my $HOST           = "my.foo.net"; # The name of the host to check
my $PROGRAM        = "host_watch";
my $LOG            = "/var/log/$PROGRAM.log"; # Our logfile
my $ACTIVE         = $FALSE; # This host is not the active host - yet
# Signals to Trap and Handle
$SIG{'INT' } = 'interrupt';
$SIG{'HUP' } = 'interrupt';
$SIG{'ABRT'} = 'interrupt';
$SIG{'QUIT'} = 'interrupt';
$SIG{'TRAP'} = 'interrupt';
$SIG{'STOP'} = 'interrupt';
$SIG{'TERM'} = 'interrupt';
```

So now that the basics are out of the way some helper functions:

```
##
# Append file: Append a string to a file.
##
sub appendfile {
    my ($fp, $msg) = @_;

    if (open(FILE, ">>$fp")) {
        print FILE (" $msg\n");
        close FILE;
    }
}

##
# Insert 0: Fix up date strings
##
sub insert0 {
    my ($date) = shift;

    if ($date < 10) {
        return "0$date";
    }

    return $date;
}

##
# Interrupt: Simple interrupt handler
##
sub interrupt {
```

```

        my $date = longfmt();
        appendfile($LOG, "$date: caught @_ exiting");
        print "caught @_ exiting\n";
        die;
    }

##
# Long format: Custom datestring for the logfile
##
sub longfmt {
    my ($sec,$min,$hour,$mday,$mon,$year,
        $yday,$yday,$iddst) = localtime(time);
    my $datestring;

    $year += 1900;
    $mon++;
    $mon = insert0($mon);
    $mday = insert0($mday);
    $min = insert0($min);
    $datestring = "$year-$mon-$mday $hour:$min";

    return($datestring);
}

```

With the helper functions out of the way (some of them should look familiar) it is time to look at two core functions. The central algorithm in this context is broken apart because there is a need for regression: that is to make absolutely sure the failure is real:

```

##
# Ping Test: Check to see if a host is alive and return
##          2 if not, 0 if yes and 3 if unknown
sub pingtest {
    my $host = shift;

    my $ping_cnt = `/bin/ping -c 8 $host | /usr/bin/wc -l`;

    if ( $ping_cnt <= 4 ) {
        return 2;
    } elsif ( $ping_cnt >= 13 ) {
        return 0;
    } else {
        return 2;
    }

    return 3;
}

##
# Regress: basically rerun the ping test just to be sure
##
sub regress {
    my $host = shift;

    my $date = longfmt();
    my $ping_result = pingtest($host); # Check again!

    if ( $ping_result >= 2 ) {
        appendfile($LOG,
            "$date: Problem with $host");
            # DO SOMETHING HERE
    } elsif ( $ping_result == 0 ) {

```

```

        appendfile($LOG,
            "$date: Missed ping on $host; now seems to be okay");
    } else {
        appendfile($LOG,
            "$date: $host is intermittent");
    }
}

```

Essentially ping is used but as the code below illustrates even if a ping check fails the regression is called just to be absolutely sure:

```

##
# MAIN: Fork and setsid().
##
unless(my $pid = fork()) {
    exit if $pid;
    setsid;
    umask 0;
    my $date = longfmt();
    appendfile($LOG, "$date: Starting $PROGRAM");
    while(1) {
        $date = longfmt();
        my $ping_result = pingtest($HOST);
        if (( $ping_result == 2 ) &&
            ($ACTIVE == $FALSE)) {
            appendfile($LOG, "$date: $HOST not replying; regressing");
            regress($HOST);
        } elsif ( $ping_result == 0 ) {
            appendfile($LOG, "$date: $HOST answered all pings");
            if ($ACTIVE == $TRUE) {
                appendfile($LOG,
                    "$date: Primary $host resumed. Stopping locally.");
                sleep 2; # Give log time to flush to disk
                # DO SOMETHING HERE
            }
        } else {
            if ($ACTIVE == $FALSE) {
                appendfile($LOG,
                    "$date: $HOST didn't answer all pings");
            } else {
                appendfile($LOG,
                    "$date: Primary down; local host is active");
            }
        }
        sleep $DELAY;
    }
}

```

Additionally if it detects a host is up; the current host stops doing whatever it was taking over.

2.1.5 Perl Program: Nagios Check System Health

Ironically there has not been as much crossover in my current environment (at least as of this writing) and my hobbyist/home coding/administration life in quite some time. For once I have found something I wrote at my job that has a direct translation into something other admins would find useful; a script that performs and reports multiple checks at once. In this the first part of the series a look at the motivation, helper functions and some core generic functions of the script.

I provision a lot of systems, a lot of them; almost 1/week (although not at that frequency - the frequency varies) - some of these systems are virtual machines, some are cloned virtual machines, some are physical servers while in the rare instance some are just devices of some type (or really dumb servers). Part of the post installation setup is adding systems to the appropriate Nagios monitors. After doing this for over a year I saw a pattern emerge - every system needed to have a set of common checks:

- Check the system time against a known good source.
- Check the load average.
- Check local disk usage.

Of course depending upon one's environment that list may need more or less.

There are many solutions to this sort of problem. One could have a config file for every host and simply drop in the new values and the config files sourced under one directory for instance. The only quick solution at the time for my configuration was to be able to wrap multiple checks into one. I looked at how I might do so within nagios and determined I was too lazy to figure out if some sort of dependency relationship could be used; since I am so lazy I looked for a plugin to do this but found it needed an agent - again being lazy I did not want to have to install agents unless there was an absolute need. The answer (as usual) became obvious - write a wrapper.

The script presented within is a first draft, there are known bugs, however, the idea behind this text is to present an idea and method for other admins to adopt so they can formulate their own similar script and be more efficient. Starting Bits

So first up some signals to trap and variables, I commented up a lot to avoid having to explain the details in the text:

```
# Signals we are interested in dealing with, the right operand is the
# subroutine which handles the given interrupt type
$SIG{'INT' } = 'interrupt';
$SIG{'HUP' } = 'interrupt';
$SIG{'ABRT'} = 'interrupt';
$SIG{'QUIT'} = 'interrupt';
$SIG{'TRAP'} = 'interrupt';
$SIG{'STOP'} = 'interrupt';
# Globals
my $USER1="/usr/local/nagios/libexec"; # Be consistent wrt Nagios
my $CHECK="HEALTH"; # the name of the check; feel free to change
my $OUTFILE = "/var/tmp/healthcheck.tmp"; # an outfile for later use
# Where we store cherry picked results; init these to a space in case they
# are not all collected
my @LOAD_VALUES = " ";
my @SYSTIME_VALUE = " ";
my @ROOTDISK_VALUE = " ";
# Default values for LOAD, ROOTDISK Usage
```

```

my $DEF_LOAD_WARN = "4,2,2";
my $DEF_LOAD_CRIT = "5,4,3";
my $DEF_DISK_WARN = 95;
my $DEF_DISK_CRIT = 98;
my $DEF_SNMP_COMMUNITY = "public";
my $STATUS = 0; # A status var to be returned to nagios
# Flags
$DNS = 1; # do check that this host has a DNS entry
$PING = 0; # don't preping by default since nagios does, switch to 1 if
           # you want to preping before bothering with the rest
# Brain dead interrupt handler
sub interrupt { # usage: interrupt \'sig\'
    my($sig) = @_;
    die $sig;
}
# Generic sub: Load a file into an array and send the array back
sub loadfile {
    my ($file) = shift;
    my @flist;
    open(FILE, $file) or die "Unable to open logfile $file: !\n";
    @flist = <FILE>;
    close FILE;
    return(@flist);
}

```

So far so good, we setup out LOAD and DISK parameters in addition to arrays to capture returned results. We are relying upon snmp checks for these but note the script could be modified to use SSH etc.

Now it is time to move onto the functions that do the work, first up is a generic return parser to construct what will be sent back to nagios:

```

# Handle results status and print a final message with values of collated data
sub check_exit { # usage: check_exit("message string",RETVAL)
    my ($msg,$ret) = @_;
    # determine our status and exit appropriately
    if ($ret >= 3) {
        print "$CHECK UNKNOWN: $msg ";
    } elsif ($ret == 2) {
        print "$CHECK CRIT: $msg ";
    } elsif ($ret == 1) {
        print "$CHECK WARN: $msg ";
    } elsif ($ret == 0) {
        print "$CHECK OK: $msg ";
    } else{
        print "$CHECK UNKNOWN STATE: $msg ";
    }
    # print what we collected - note if one fails we do not collect the rest
    chomp (@SYSTIMEVALUE);
    chomp (@LOAD_VALUES);
    print("@SYSTIME_VALUE, System Load @LOAD_VALUES, Rootdisk @ROOTDISK_VALUE");
    unlink($OUTFILE); # delete the temp file for good
    exit ($ret);      # exit appropriately so nagios knows what to do
}

```

The exit function uses the return number to determine the warning level (if any) and passes along an optional message string. Note that regardless of which check failed the function returns all available data. The idea was (when it was written) if disk is low it might be causing a high load etc. The next function greps CRITICAL or WARN from the output file; this is because the script actually calls other Nagios checks which will leave the

status string in the output file:

```
# Check the outfile in some cases for a SNMP warn or critical
# send back the appropriate signal for nagios
sub check_outfile { # usage: check_outfile
    my @critical = `grep CRITICAL $OUTFILE`;
    if (@critical) {
        return 2;
    }
    my @warn = `grep WARN $OUTFILE`;
    if (@warn) {
        return 1;
    }

    return 0;
}
```

Next is the usage, note that not all of the capabilities have been scripted yet so this is a look ahead (kind of) at the next text:

```
# ye olde usage message
sub usage {
    print "Usage: $0 [-u[-H ||[ -lw -lc -dw -dc ]]\n";
    print "Usage: $0 [--nodns][--noping][--snmp \"community [user] [pass]\"\n";
    print "Options:\n";
    print " -H      Check system called (required)\n";
    print " -lw     Set load warning values\n";
    print "         Default: $DEF_LOAD_WARN\n";
    print " -lc     Set load critical values\n";
    print "         Default: $DEF_LOAD_CRIT\n";
    print " -dw     Set rootdisk warning percent\n";
    print "         Default: $DEF_DISK_WARN\n";
    print " -dc     Set rootdisk critical percent\n";
    print "         Default: $DEF_DISK_CRIT\n";
    print " --nodns Do not check for DNS resolution\n";
    print " --noping Do not preping to make sure the host is up\n";
    print "         Note: this will improve performance\n";
    print " --snmp  Set SNMP community name\n";
    print "         Default: $DEF_SNMP_COMMUNITY\n";
    print " -u      Print usage message and exit\n";
}
```

Using the usage message as a roadmap the first check is the load check. For this the script simply calls the existing `check_snmp nagios` check, note the `snmp community` is an argument:

```
# Check load
sub load { # usage: load($host_or_ip,warn,critical,community)
    my ($host,$warn,$crit,$comm) = @_;
    system("$USER1/checksnmp -H $host -C $comm -o \
        .1.3.6.1.4.1.2021.10.1.3.1,.1.3.6.1.4.1.2021.10.1.3.2,\
        .1.3.6.1.4.1.2021.10.1.3.3 -w $warn -c $crit \
        -l \"Load 1min/5min/10min\" > $OUTFILE");
    my $r = check_outfile();
    @LOAD_VALUES = `cat $OUTFILE|`;
    awk '{ print \"$3 \" \" \" \"$5 \" \" \" \"$6 \" \" \" \"$7}';
    if ($r > 0) {
        if ($STATUS < $r) {
            $STATUS = $r;
        }
    }
}
```

```
}
```

The function gets the values, stores and finally checks their status. Next in order is a simple one, using snmp again check the root filesystem:

```
# Check rootdisk
sub rootdisk { # usage: rootdisk(host_or_ip,warn,crit,community)
    my ($host,$warn,$crit,$comm) = @_;
    system("$USER1/checksnmp -H $host -C $comm \
-o 1.3.6.1.4.1.2021.9.1.9.1,.1.3.6.1.4.1.2021.9.1.7.1,\
.1.3.6.1.4.1.2021.9.1.8.1,.1.3.6.1.4.1.2021.9.1.3.1,\
.1.3.6.1.4.1.2021.9.1.2.1 -w $warn -c $crit > $OUTFILE");
    my $r = check_outfile();
    @ROOTDISK_VALUE = `cat $OUTFILE|\
    awk '{print \$4 \" \" \$5 \" \" \$6}'`;
    if ($r > 0) {
        if ($STATUS < $r) {
            $STATUS = $r;
        }
    }
}
}
```

With the core checks out of the way it is time for the main loop:

```
# MAIN
# init our default values; then parse input to see if we want to change any
my $load_warn = $DEF_LOAD_WARN;
my $load_crit = $DEF_LOAD_CRIT;
my $disk_warn = $DEF_DISK_WARN;
my $disk_crit = $DEF_DISK_CRIT;
my $snmp_community = $DEF_SNMP_COMMUNITY;
my $host;
while ( my $i = shift @ARGV ) {
    if ($i eq '-u') {
        usage();
        exit (0);
    } elsif ($i eq '-H') {
        $host = shift @ARGV;
    } elsif ($i eq '-lw') {
        $load_warn = shift @ARGV;
    } elsif ($i eq '-lc') {
        $load_crit = shift @ARGV;
    } elsif ($i eq '-dw') {
        $disk_warn = shift @ARGV;
    } elsif ($i eq '-dc') {
        $disk_crit = shift @ARGV;
    } elsif ($i eq '--nodns') {
        $DNS = 0;
    } elsif ($i eq '--ping') {
        $PING = 1;
    } elsif ($i eq '--snmp') {
        $snmp_community = shift @ARGV;
    }
}
# there is no spoon...
if (!$host) {
    print "Error: no host specified\n";
    usage();
    exit (1);
}
# if we wanna ping go ahead XXX-jrf do we care about stats?
```

```
if ($PING == 1) {
    preflight($host);
}
# if we wanna resolve then resolve
if ($DNS == 1) {
    dns($host);
}
# Call checks
load($host,$load_warn, $load_crit,$snmp_community);
system($host,$snmp_community);
rootdisk($host,$disk_warn,$disk_crit,$snmp_community);
# were all good - go ahead and exit
check_exit ("",$STATUS);
```

2.2.0 The C Programming Language

The C programming level is arguably the most popular systems programming language available to date. It was invented by UNIX pioneers Brian Kernighan and Dennis Ritchie with the goal of being easily ported from hardware platform to hardware platform while still being efficient. In programming terms it can be viewed as being a step above assembly language but (and slightly so) below programming languages that have built in object oriented pragmas.

The C language is also strongly typed; in the previous languages all variables and data structure datas have been implied by context. In C types are declared with the variables so the compiler knows how to treat them.

The C language also requires a compilation stage whereas in Perl and the shell one simply makes the source file an executable, a C program must be built first then the resulting executable object can be executed. C also depends on a large set of libraries using the include directive. Lastly; C also has strong MACRO creation capabilities. This text will not cover the details of includes and MACROS: it is assumed the reader already has a basic understanding of compiling and executing a C program.

As a quick refresher however, following is a classic hello world program save in the hello_world.c file:

```
#include <stdio.h>
```

```
int main (void)
{
    printf("Hello world\n");
    return 0;
}
```

To compile then execute simply:

```
cc hello_world.c -o hello
./hello
Hello world
```

2.2.1 Small C Programs

Not unlike any programming language the size and power of C programs is not relevant to the amount of code required to create a usable program. So once again the jump off will be small but pretty potent little programs. The difference in C versus previous languages is that C can make excellent use of libraries available.

2.2.1.i Print User Information

4.4-BSD and Net/2-BSD based systems come with a great `userinfo` program that prints out the GECOS field of a user in a terminal. Unfortunately not all Unix systems ship this program. Time to write one with the ability to specify multiple users. First up the included header files from the system's standard library and a defined constant:

```
#include <grp.h>          /* Header file with group info */
#include <pwd.h>          /* The password db header */

#include <sys/types.h>    /* System internal types */

/* The rest are for printing etc. */
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#define PACKAGE "pwuser" /* The name of the program */
```

Easy enough, now onto prototypes. In C we need to make sure the parts of the program that need to know about other functions in fact can. Declaring prototypes also provides a sort of road map to how a particular program is constructed:

```
void getuserinfo (char *username); /* Get and print user info */
void usage (void);                /* In case someone needs it */
```

Now it is time for the central algorithm of the program, getting and printing the user information. The program leverages the standard library functions for accessing system databases and assigns them to another standard library data structure; using the data structure it then prints out the data:

```
/* Since we do the print here we do not need to return anything */
void getuserinfo (char *username)
{
    struct passwd *user_pwd_info; /* The pwd-db entries */
    struct group  *user_grp_info; /* The group entries */
    char          **user_members; /* List of groups */

    user_pwd_info = getpwnam(username); /* Access the username */

    /* Ooops - couldn't find them */
    if (!user_pwd_info) {
        printf("Could not find info about user %s\n", username);
        return;
    }

    /* Get group info */
    user_grp_info = getgrgid(user_pwd_info->pw_gid);
```

```

        /* Now print it all out in one shot */
printf("UserID: %d\n"
      "GroupID: %d\n"
      "Username: %s\n"
      "Default Group: %s\n"
      "Home Directory: %s\n"
      "Default Login Shell: %s\n"
      "Misc Information: %s\n",
      user_pwd_info->pw_uid, user_pwd_info->pw_gid,
      user_pwd_info->pw_name, user_grp_info->gr_name,
      user_pwd_info->pw_dir, user_pwd_info->pw_shell,
      user_pwd_info->pw_gecos);
}

```

It looks daunting but truly isn't. The easiest of the two functions is the usage print.

```

void usage (void)
{
    printf( PACKAGE " [user1 user2 user3...]\n"
          PACKAGE " usage\n"
          );
}

```

Not too difficult at all. Lastly is the main function - as per the norm it is heavily commented for instructional use:

```

/* We use argc as the array counter and argv is the array of
   possible usernames requested */
int main (int argc, char *argv[])
{
    int c; /* Index counter for looping over the names */

    /* If someone forgot usernames */
    if (argc <= 1) {
        printf("Syntax error\n"); /* Oopsee */
        usage(); /* Print the usage message */
        return 1; /* let the shell know something is wrong */
    }

    /* If someone didn't know how to use the program */
    if (strcmp(argv[1], "usage") == 0) {
        usage(); /* Print the usage message */
        return 0; /* They wanted to know so exit 0 */
    }

    /* Loop over the names; call getuserinfo for each one */
    for (c = 1; c < argc; c++) {
        getuserinfo(argv[c]);

        /* Tack on an extra line in between names */
        if (argv[c + 1])
            printf("\n");
    }

    return 0; /* We did good - exit 0 */
}

```

2.2.1.ii etu v 0.0.1

As stated in the beginning of the C section, the C language can make excellent use of programming libraries. One excellent set of libraries comes from the enlightenment project (<http://enlightenment.org/>). The enlightenment project has a jpeg thumbnailing library called epeg. In the most basic form epeg can take a jpeg file and rescale it (generally smaller). In the following ethumb.c program an extremely simple example of creating a thumbnailer in less than 21 lines of code:

```
#include "Epeg.h" /* This has to be installed ! */

int main(int argc, char **argv) /* Point to the argv array this time */
{
    Epeg_Image *image; /* An empty image instance */

    if (argc != 3) { /* There must be 3 inputs described below */
        printf("Usage: %s input.jpg thumb.jpg\n", argv[0]);
        return 1; /* Oops */
    }

    image = epeg_file_open(argv[1]); /* Call epeg to open the file */

    if (!image) { /* If no image error with a -1 */
        printf("Cannot open %s\n", argv[1]);
        exit(-1);
    }

    epeg_decode_size_set      (image, 128, 96); /* WidthxHeight */
    epeg_quality_set          (image, 75);     /* Quality */
    epeg_file_output_set      (image, argv[2]); /* Set the new name */
    epeg_encode                (image);        /* Encode to new */
    epeg_close                 (image);        /* Close */

    return 0; /* All conditions normal ... */
}
```

Pretty easy! Of course the enlightenment libraries and epeg libs in particular need to be installed in order for it to work. It is easy to see how the ethumb program could be expanded - and it is later on in this section.

2.2.2 File and I/O Examples in C

Reading from and writing to files in C is actually very similar to Perl (or is that the other way around?) with the exception of typing. In the next two examples; only reading of system information is done and reformatted out to the screen.

2.2.2.i lscpu

Linux systems come with a very nice interface which provides a great deal of information about a system using a pure file interface instead of calling system call APIs or using the BSD sysctl interface: a virtual filesystem call /proc which along with process information contains hardware and kernel data. Both of the example programs utilize /proc by reading data out of it and reformatting it for easier (convenient) use. The first of these programs is called lscpu which does exactly what it sounds like:

```
./lscpu
Processor Information for argos
OS: Linux version 2.6.26-1-amd64
CPU 0 is Processor Type: AuthenticAMD   AMD
Processor Speed in MHz: 1000.000
Processor Cache Size: 512
Processor Speed in Bogomips: 2011.10
CPU 1 is Processor Type: AuthenticAMD   AMD
Processor Speed in MHz: 1000.000
Processor Cache Size: 512
Processor Speed in Bogomips: 2011.10
RTC Current Time: 21:09:02      RTC Date: 2009-01-10
RTC Periodic Frequency: 2048   RTC Battery Status: okay
```

lscpu lists out cpu, OS and real time clock information. Believe it or not it takes a lot of reading to get this information. First a look at one of the smaller functions that reads some information then a full program listing will follow. Here are the headers and definitions:

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAXLEN 1024

/* the file handles we will be using */
static char *pfh[]=
{
    "/proc/cpuinfo",
    "/proc/sys/kernel/hostname",
    "/proc/sys/kernel/ostype",
    "/proc/sys/kernel/osrelease",
    "/proc/driver/rtc",
};

/* Function Prototypes */
void error_output_mesg(char *locale);
void get_hostname_info(void);
void get_ostype_info(void);
void get_osrelease_info(void);
void get_cpu_info(void);
void get_rtc_info(void);
```

A lot more functions than dealt with so far. Here is the error output function:

```
void error_output_mesg(char *locale)
{
    fprintf(stderr, ("%s: %s\n"), locale, strerror(errno));
    exit (1);
}
```

Next a detailed look at the hostname get function, the rest of the functions all work similar to this one except they are scanning in different parts of different files. The pfh structure is used relative to each file needed in each function:

```
void get_hostname_info()
{
    FILE *hostfp; /* A filehandle for the host info file */
    static char hostch[MAXLEN]; /* A buffer for the file */
    static char hostname[MAXLEN]; /* The resulting hostname */

    /* Try to open the file or fail */
    if((hostfp = fopen(pfh[1], "r")) == NULL)
        error_output_mesg(pfh[1]);

    /* While the file is open scan in the hostname field from it
    then assign it to the hostname variable; finally print it out */
    while(fgets(hostch, MAXLEN, hostfp) != NULL) {
        sscanf(hostch, "%s", hostname);
        printf("Processor Information for %s\n", hostname);
    }

    fclose(hostfp);
}
```

Now for the rest of the helper functions. Note that they are all similar to get_hostname() just gathering up and printing more information:

```
void get_ostype_info()
{
    FILE *osfp;
    static char osch[MAXLEN];
    static char ostype[MAXLEN];

    if((osfp = fopen(pfh[2], "r")) == NULL)
        error_output_mesg(pfh[2]);

    while(fgets(osch, MAXLEN, osfp) != NULL) {
        sscanf(osch, "%s", ostype);
        printf("OS: %s", ostype);
    }

    fclose(osfp);
}

void get_osrelease_info()
{
    FILE *osrfp;
    static char osrch[MAXLEN];
    static char osrelease[MAXLEN];

    if((osrfp = fopen(pfh[3], "r")) == NULL)
        error_output_mesg(pfh[3]);
}
```

```

while(fgets(osrch, MAXLEN, osrftp) != NULL) {
    sscanf(osrch, "%s", osrelease);
    printf(" version %s\n", osrelease);
}

fclose(osrftp);
}

void get_cpu_info()
{
    FILE *cpufp;
    static char ch[MAXLEN];
    char line[MAXLEN];

    if ((cpufp = fopen(pfh[0], "r")) == NULL)
        error_output_mesg(pfh[0]);

    while (fgets(ch, MAXLEN, cpufp) != NULL) {
        if (!strncmp(ch, "processor", 9)) {
            sscanf(ch, "%*s %*s %s", line);
            printf("CPU %s", line);
        } else if (!strncmp(ch, "vendor_id", 9)) {
            sscanf(ch, "%*s %*s %s", line);
            printf(" is Processor Type: %s ", line);
        } else if (!strncmp(ch, "model name", 10)) {
            sscanf(ch, "%*s %*s %*s %s", line);
            printf(" %s\n", line);
        } else if (!strncmp(ch, "cpu MHz", 7)) {
            sscanf(ch, "%*s %*s %*s %s", line);
            printf("Processor Speed in MHz: %s\n", line);
        } else if (!strncmp(ch, "cache size", 10)) {
            sscanf(ch, "%*s %*s %*s %s", line);
            printf("Processor Cache Size: %s\n", line);
        } else if (!strncmp(ch, "bogomips", 8)) {
            sscanf(ch, "%*s %*s %s", line);
            printf("Processor Speed in Bogomips: %s\n", line);
        }
    }
    fclose(cpufp);
}

void get_rtc_info()
{
    FILE *rtcfp;
    static char ch[MAXLEN];
    char line[MAXLEN];

    if ((rtcfp = fopen(pfh[4], "r")) == NULL)
        error_output_mesg(pfh[4]);

    /* Just grab stuff from a certain position and dump it to stdout */
    while (fgets(ch, MAXLEN, rtcfp) != NULL) {
        if (!strncmp(ch, "rtc_time", 8)) {
            sscanf(ch, "%*s %*s %s", line);
            printf("RTC Current Time: %s\t", line);
        } else if (!strncmp(ch, "rtc_date", 8)) {
            sscanf(ch, "%*s %*s %s", line);
            printf("RTC Date: %s\n", line);
        } else if (!strncmp(ch, "periodic_freq", 13)) {
            sscanf(ch, "%*s %*s %s", line);
            printf("RTC Periodic Frequency: %s\t", line);
        }
    }
}

```

```

        } else if (!strncmp(ch, "batt_status", 11)) {
            sscanf(ch, "%*s %*s %s", line);
            printf("RTC Battery Status: %s\n", line);
        }
    }
    fclose(rtcfp);
}

```

Ironically with all of the functions doing so much work and not needing to return anything the main() function ends up looking pretty boring:

```

int main(void)
{
    get_hostname_info();
    get_ostype_info();
    get_osrelease_info();
    get_cpu_info();
    get_rtc_info();

    printf("\n");

    return 0;
}

```

So why use this program when awk or Perl would do just as well? Good question, the program was authored when CPU cycles were still somewhat expensive so C was the choice at the time, nowadays a good shell script can do the job just as well.

2.2.2.ii mmw

Micro Memory Watcher or mmw is a clone of the free program that is slightly prettier. Mmw has a lot more complexity than the lscpu program. In addition to reading and printing some data from /proc it also:

- Can poll periodically
- Converts raw bytes into MB, GB and TB
- Has various control switches
- Can also look at swap usage

First up in mmw the includes, definitions, prototypes and helper functions:

```

#include <ctype.h>
#include <getopt.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>

#define MAXLEN 256
#define PROGRAM "mmw"
#define VERSION "1.9"
#define MEMINFO "/proc/meminfo"

void version(void);
void usage(void);
void print_head(unsigned long int fws, char * units, unsigned int dsize);
void read_meminfo(unsigned int polls, unsigned int interval,

```

```

        unsigned int sf, unsigned int hflag, unsigned int dsize);

/* simple reusable version print */
void version() { printf("%s %s\n", PROGRAM, VERSION); }

/* reusable usage print */
void usage()
{
    printf("usage: %s [options args]\n", PROGRAM);
    printf("usage: %s [-h|--human] [-i|--interval] [-p|--polls POLLS]\n", PROGRAM);
    printf("    [-u|--usage] [-v|--version]\n");
    printf("options:\n");
    printf("  -h|--human           Human readable format.\n");
    printf("  -i|--interval SECONDS Seconds between polls.\n");
    printf("  -p|--polls      NPOLLS Times to poll.\n");
    printf("  -s|--swap       Poll swap information as well.\n");
    printf("  -u|--usage      Print usage message.\n");
    printf("  -v|--version    Print version and exit.\n");
}

```

The first part of the source file should all seem pretty familiar. In order to illustrate what the information is, for mmw a tabular format is used since it can poll (similar to vmstat) so a header is needed. Because the data can be of variable size an if/else ladder determines the print width of the fields:

```

void print_head(unsigned long int fws, char * units, unsigned int dsize)
{
    int count;

    static char *header[]=
    {
        "total",
        "free",
        "shared",
        "buffer",
        "cached",
        "swap",
        "sfree",
    };

    printf("Memory Usage in: %s\n", units);

    /* determine the field width for the header */
    if(fws <= 100000) {
        for(count = 0; count <= dsize; count++)
            printf("%-8s", header[count]);
    } else if(fws <= 100000000) {
        for(count = 0; count <= dsize; count++)
            printf("%-11s", header[count]);
    } else if(fws > 100000000) {
        for(count = 0; count <= dsize; count++)
            printf("%-14s", header[count]);
    } else {
        for(count = 0; count <= dsize; count++)
            printf("%-15s", header[count]);
    }

    printf("\n");
}

```

Since the core algorithm is more complex than previous examples it is being saved for last, instead, now a

jump into how the main() function looks. There are some constructs here worth study:

- A getopt data structure is introduced. This contains information about how the command line switches should look and prepares them for parsing.
- A switch/case which iterates over switches and handles them.
- Many many flags. In the old days bitfields were often used to contain flags; we do not need to do that so regular old integers will do the job.

There is a lot of commenting to help in this larger than usual main():

```
int main(int argc, char *argv[])
{
    /*
     * c = index for switch and getopt
     * hflag = Human readable format?
     * dsize = default field width size
     * poll = how many times to poll
     * interval = interval in seconds to poll
     */
    int c, hflag, interval, poll, dsize;

    /* Defaults */
    interval = 5;
    poll      = 5;
    hflag     = 0;
    dsize     = 4;

    /* If no arguments are specified do one pass and exit */
    if(argc == 1) {
        read_meminfo(poll, interval, 1, hflag, dsize);
        return 0;
    }

    /*
     * Now we do a parsing loop using the GNU getopt capability
     * A structure is setup containing names of the switches
     * and their short letter versions.
     */
    while (1) {
        static struct option long_options[] =
            {
                {"human",      no_argument,      0, 'h' },
                {"interval",   required_argument, 0, 'i' },
                {"poll",       required_argument, 0, 'p' },
                {"swap",       no_argument,      0, 's' },
                {"version",    no_argument,      0, 'v' },
                {"usage",      no_argument,      0, 'u' },
                {0,0,0,0} /* This is a filler for -1 */
            };

        int option_index = 0; /* the option index counter */

        /* call getopt long to fill out which options are being used */
        c = getopt_long (argc, argv, "hi:p:svu", long_options, &option_index);

        if (c == -1) break; /* break out when we have counted down */

        switch(c) {
            case 'h':
                hflag = 1;

```

```

        break;
    case 'i':
        if (isalpha(*optarg)) { /* make sure it is a number! */
            fprintf(stderr, "Error: interval must be a number\n");
            usage();
            exit (2);
        }
        interval = atoi(optarg);
        break;
    case 'p':
        if (isalpha(*optarg)) {
            fprintf(stderr, "Error: poll must be a number\n");
            usage();
            exit (2);
        }
        poll = atoi(optarg);
        break;
    case 's':
        /* If we want swap then the dsize needs to be bigger */
        dsize = (dsize + 2);
        break;
    case 'v':
        version();
        return 0;
        break;
    case 'u':
        usage();
        return 0;
        break;
    default:
        usage();
        exit (2);
        break;
}
}

/* Run the read_meminfo callback function - recurse until done */
read_meminfo(poll, interval, 1, hflag, dsize);

return 0;
}

```

Again; looks like a lot but upon closer examination the code is pretty clear. Time for the fun (okay maybe not so fun) part of the core algorithm. Read meminfo does what it says, it reads and reformats data from /proc/meminfo but it does so recursively. Essentially it will call itself until the number of polls it passes to itself has elapsed. This is nothing more than clever programming and saved the author the hassle of having to write loops in other parts of the program. In this particular program recursion saves programmer time and system time, however, often recursion can be leveraged to do both.

The core algorithm is pretty big and needs a lot of setup so instead of the usual print it and comment it is broken up into several chunks with commenting and discussion points after each section:

```

/* recursively prints out mem information */
void read_meminfo(unsigned int polls, unsigned int interval,
                 unsigned int sf, unsigned int hflag, unsigned int dsize)
{
    FILE *fp; /* The file handle for this program */
    static char ch[MAXLEN]; /* Character buffer */
    unsigned long int mem_array[7]; /* Data structure to hold meminfo */
    unsigned int count; /* A counter */
}

```

```

long int hdiv;                /* Used for division                */
char * units;                /* The units being printed    */
short int i;                 /* Positional number          */

units = "kB"; /* The default units from the file */
hdiv = 1;     /* Default value */

/* initialize the array */
for (i = 0; i <= (dsize + 1); i++)
    mem_array[i] = 0;

if(polls != 0) {
    sleep(interval); /* Go ahead and sleep unless done */

        /* make sure we can read the file! */
    if((fp = fopen(MEMINFO, "r")) == NULL)
        fprintf(stderr, "could not open file %s\n", MEMINFO);
}

```

Note that there is a lot of information being sent to the function. Also unlike previous functions there is a lot of setup involved. Next the mem_array data structure is populated using data from the /proc/meminfo file in a similar fashion to getting information for lscpu:

```

while(fgets(ch, MAXLEN, fp) != NULL) {
    if(!strncmp(ch, "MemTotal:", 9)) {
        sscanf(ch + 10, "%lu", &mem_array[0]);
    } else if(!strncmp(ch, "MemFree:", 8)) {
        sscanf(ch + 10, "%lu", &mem_array[1]);
    } else if (!strncmp(ch, "MemShared:", 10)) {
        sscanf(ch + 10, "%lu", &mem_array[2]);
    } else if(!strncmp(ch, "Buffers:", 8)) {
        sscanf(ch + 10, "%lu", &mem_array[3]);
    } else if(!strncmp(ch, "Cached:", 7)) {
        sscanf(ch + 10, "%lu", &mem_array[4]);
    } else if((dsize > 4) && (!strncmp(ch, "SwapTotal", 9))) {
        sscanf(ch + 10, "%lu", &mem_array[5]);
    } else if((dsize > 4) && (!strncmp(ch, "SwapFree", 8))) {
        sscanf(ch + 10, "%lu", &mem_array[6]);
    }
}

fclose(fp); /* all done - close /proc/meminfo */

```

The next chunk of code deals with the human readable flag; the program must determine the units to print in by easily looking at the value of all the memory. Note it also resets the divisor parameter to be used later on in field width parameters:

```

if (hflag) {
    if(mem_array[0] <= 999) {
        hdiv = 1;
    } else if(mem_array[0] <= 999999) {
        hdiv = 1000;
        units = "MB";
    } else if(mem_array[0] <= 999999999) {
        hdiv = (1000000);
        units = "GB";
    } else {
        hdiv = (1000000000);
        units = "TB";
    }
}

```

Time for the last piece of the core algorithm; determine the field width needed and print:

```
/* If this is the first line print the header */
if(sf == 1) print_head(mem_array[0]/hdiv, units, dsize);

/* determine the field width for each printout */
if(mem_array[0]/hdiv <= 100000) {
    for(count = 0; count <= dsize; count++)
        printf("%-8li", mem_array[count]/hdiv);
} else if(mem_array[0]/hdiv <= 100000000) {
    for(count = 0; count <= dsize; count++)
        printf("%-11li", mem_array[count]/hdiv);
} else if(mem_array[0]/hdiv > 100000000) {
    for(count = 0; count <= dsize; count++)
        printf("%-14li", mem_array[count]/hdiv);
} else {
    for(count = 0; count <= dsize; count++)
        printf("%-15li", mem_array[count]/hdiv);
}

printf("\n");
```

Finally - call ourself and close the function out:

```
        read_meminfo(polls - 1, interval, 0, hflag, dsize);
    }
}
```

Here is some sample output of mmw:

```
[19:12:33 jrf@argos:~/src/mmw]$ ./mmw 5 5
Memory Usage in: kB
total    free      shared    buffer    cached
4030544  41724     0         98432     3120940
4030544  41740     0         98440     3120960
4030544  41756     0         98448     3120968
4030544  41748     0         98452     3120964
4030544  41872     0         98460     3120952
[19:13:10 jrf@argos:~/src/mmw]$ ./mmw -h -i 5 -p 3
Memory Usage in: GB
total    free      shared    buffer    cached
4        0         0         0         3
4        0         0         0         3
4        0         0         0         3
```

2.2.3 Networking and OS C Programs

For the systems and networking programs it is time to take a look at the level of detail the C program language can get a programmer to; this is especially true on BSD, Linux and Unix systems since C was invented both on and for Unix.

2.2.3.i Making Forks

The various system calls on Unix and Unix-like systems can provide rich detailed information about how an Operating System is performing. The following example is a very small program that is designed to create a false load using forks. It was part of the 4.4 BSD Operating System curriculum when it was still taught at colleges and corporate campuses in the 1990s.

```
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(argc, argv)
    int argc;
    char *argv[];
{
    register int nforks, i;
    char *cp;
    int pid, child, status, brksize;

    if (argc < 2) { /* If not enough arguments - bail */
        printf("usage: %s number-of-forks sbrk-size\n", argv[0]);
        exit(1);
    }

    nforks = atoi(argv[1]); /* Check that number of forks will work */
    if (nforks < 0) {
        printf("%s: bad number of forks\n", argv[1]);
        exit(3);
    }

    brksize = atoi(argv[2]); /* Check the break size */
    if (brksize < 0) {
        printf("%s: bad size to sbrk\n", argv[2]);
        exit(3);
    }

    cp = (char *)sbrk(brksize); /* Setup cp */
    if ((int)cp == -1) {
        perror("sbrk");
        exit(4);
    }

    for (i = 0; i < brksize; i += 1024)
        cp[i] = i;

    while (nforks-- > 0) { /* Spin through fork generation */
        child = fork();
        if (child == -1) {
            perror("fork");
            exit(-1);
        }
    }
}
```

```

    }
    if (child == 0)
        exit(-1);
    while ((pid = wait(&status)) != -1 && pid != child)
        ;
}

return 0;
}

```

Note the succinctness of the forks generation program. it is small, compact, simple, does not have a great deal of dependencies but is incredibly powerful and has the potential to even be somewhat dangerous.

2.2.3.ii A Tiny Packet Sniffer

Converse to the tiny forks program is a powerful tool indeed, a packet sniffer that utilizes the libpcap library (<http://tcpdump.org/>). In a more full fledged sniffer there would be a great deal more work; but in the smallest form a sniffer can actually be even smaller than the lscpu program. First the header files and another callback function:

```

#include <pcap.h>
#include <string.h>
#include <stdlib.h>

#define MAXBYTECAP 2048 /* The maximum bytes to capture - not packets */

void pcap_callback(u_char *arg, const struct pcap_pkthdr * pkthdr,
                  const * packet)
}

    int i = 0;
    int * counter = (int *) arg; /* typecast arg as counter */

    printf("Packet Count: %d\n", ++(*counter));
    printf("Received Packet Size: %d\n", pkthdr->len);
    printf("Payload:\n");
    for (i = 0; i < pkthdr->len; i++) {
        if (printf("%c ", packet[i]);
    else
        printf(". ");

    if ((i%16 == 0 && i != 0) || i == pkthdr-> len - 1)
        printf("\n");
    }

    return;
}

```

There are some interesting constructs in the above code. Note that the key is the function is compact and does just what it says - prints out packet data only. Of course it could be massively expanded to include all sorts of information. Also note for brevity there are no prototypes. Sometimes if a program is small enough prototypes are not needed: the code is self explanatory since it is such a small amount. Now the main() function:

```

int main()
{
    int i = 0, count = 0;
    pcap *descr = NULL;
    char errbuf(PCAP_ERRBUF_SIZE), *device = NULL;

```

```
    memset(errbuf, 0, PCAP_ERRBUF_SIZE);

    device = pcap_lookupdev(errbuf);
    printf("Opening device: %s\n", device);
    descr = pcap_open_live(device, MAXBYTECAP, 1, 512, errbuf);
    pcap_loop(descr, -1, pcap_callback, (u_char *)&count);

    return 0;
}
```

If it is not clear, the count controls the number of reads. The callback function runs continuously due to the -1. To see how the pcap functions work if libpcap (and the development files) are installed on your system simply type

`man pcap`

And read on - there is a great deal one can do with libpcap!

2.2.4 C Program: The Enlightenment Thumbnailing Utility

The Enlightenment Thumbnailing Utility or the short version - etu - is a more fleshed out version of the earlier ethumb program. In this program there is a separate header file and compact well contained functions. Additionally this fuller fledged program can do other image types by utilizing the imlib2 image rendering library. First up the header file:

```
#ifndef ETUH /* We need to use this in case later we add other source */
#define ETUH /* files to the program and call the header many times */

#include <errno.h>
#include <getopt.h>
#include <unistd.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <argp.h>

#include <sys/types.h>
#include <sys/stat.h>

#include <Imlib2.h>

#include <Epeg.h>

#define PACKAGE "etu"

/*
 * Prototypes - if we were to add other source files we might want
 * to move some of these into the source file where they are used
 * only. Since we only have one source file though we are putting them
 * here to make our roadmap
 */
void update_image      (char *, char *, int, int, int);
void update_rescaled  (char *, char *);
void scale_peg        (char *, char *, int, int, int);
void scale_imlib2     (char *, char *, int, int, int);
void usage            (void);
int  check_handle     (char *);
char *fullpath        (char *, char *);
char *gettype         (char *);

#endif /* We close off our compile time check */
```

Note the comment about prototypes - it is a very true statement - you should not make available any more information about a program than need be. In the case of etu, since it consists of its build header file and program file showing the prototypes is perfectly acceptable. If one added other source files to the program then it might make more sense to move prototypes into source files that they are exclusively used by and put only shared prototypes in the common header files.

Now onto the main program which even though it does a great deal is not nearly as daunting as it appears to be. Instead of following the order of functions a look at functions by type to gear up towards the more complex ones is a better approach. First up are some utility functions - they are:

- `check_handle`: Make sure a handle exists
- `fullpath`: establish a full real path based on input
- `gettype`: get an image type

```

int check_handle(char *dir)
{
    struct stat statbuf; /* using gnu libs we make sure it is there */

    if (stat(dir, &statbuf) != 0)
        return 1;

    return 0;
}

char *fullpath(char *file, char *dir) /* Establish a full path */
{
    static char path[PATH_MAX];

    strcpy(path, dir);
    strcat(path, "/");
    strcat(path, file);

    return (path);
}

char *gettype(char *img_src) /* here we leverage imlib to get the type */
{
    char *image;
    char *format;

    image = imlib_load_image(img_src);
    imlib_context_set_image(image);
    if ((format = imlib_image_format()) == NULL) {
        fprintf(stderr, "Internal error\n");
        return (NULL);
    }

    imlib_free_image();
    return (format);
}

```

Those three functions can be considered utility functions or helper functions. They probably could have been somewhere else or even combined but for simplicity they are factored down to the simplest form to allow the program core to be easily read and manipulated by a programmer. The next function is the usage message for the program:

```

void usage(void)
{
    printf( PACKAGE " [options][arguments]\n"
           PACKAGE " [-D|--daemonize interval]\n"
           PACKAGE " [-d|--dir dir][-s|--src][-h|--height height]\n"
           PACKAGE " [-w|--width width][-q|--quality percent]\n"
           PACKAGE " [-f|--file filename][-o|--output filename]\n"
           "Single file Options:\n"
           " -f|--file file Single input image filename\n"
           " -o|--output file Output image filename\n"
           "Directory Options:\n"
           " -s|--src dir Original images directory\n"
           " -d|--dir dir Output directory\n"
           "Global Options:\n"

```

```

        " -h|--height size Height in pixels default: 96px\n"
        " -q|--quality level Quality level (1-100) default: 75%\n"
        " -w|--width size Width in pixels default: 128px\n"
    );
}

```

Straightforward atomic print. Now onto a set of algorithms. Unlike most of the other C programs in this one there is a reliance upon a set of fall through checks that decide exactly which rendering engine to choose. If the image is a jpeg then call out epeg otherwise call out imlib2 - the catch is if the image has already been scaled in the destination directory then skip it. The program also needs to determine if it is a file or directory: which leads to jumping (awkwardly) to the main() function which is broken up - first the headers, declaration and all of the local variables with their defaults:

```
#include "etu.h"
```

```

int main(int argc, char **argv)
{
    int c; /* opt counter */
    int interval; /* interval for daemon mode */
    int dst_height; /* height of the destination image(s) */
    int dst_quality; /* quality of the destination image(s) */
    int dst_width; /* width of the destination image(s) */
    char *dst_dp; /* The directory that new images will be in */
    char *src_dp; /* Where the source images live */
    char *src_fp; /* Individual handle for one input */
    char *dst_fp; /* Individual file handle for output */
    char *type; /* Image type by string */

    /* Defaults */
    interval = 0;
    src_fp = NULL;
    dst_fp = NULL;
    src_dp = NULL;
    dst_dp = NULL;
    dst_height = 96;
    dst_quality = 75;
    dst_width = 128;
    type = NULL;
}

```

Look at all that stuff... the program has the capability of either rescaling a single image or an entire source directory to destination directory of images. The catch is if it is doing source to destination images it wants to check to see if the image is already in the destination - why? because that means it can also be used as a thumbnailing cache engine that runs right from the command line. Now onto the options parser - again GNU getoptlong() and associated data structures are used:

```

/*
 * Options parsing:
 * d -dir Destination directory for a set of input images
 * h -height Height of the destination image(s)
 * f -file Input image file
 * o -output Output file
 * q -quality Set the quality of the destination image(s)
 * s -src Source directory of original files
 * w -width Width of the destination image(s)
 */
while (1) {
    static struct option long_options[] = {
        {"daemon", required_argument, 0, 'D'},
    };
}

```

```

        {"dir",      required_argument, 0, 'd'},
        {"height",  required_argument, 0, 'h'},
        {"file",    required_argument, 0, 'f'},
        {"output",  required_argument, 0, 'o'},
        {"quality", required_argument, 0, 'q'},
        {"src",     required_argument, 0, 's'},
        {"width",   required_argument, 0, 'w'},
        {0,0,0,0}
};

int option_index = 0;

c = getopt_long (argc, argv, "D:d:f:h:o:q:s:w:",
                long_options, &option_index);

if (c == -1)
    break;

switch (c) {
    case 'D':
        interval = atoi(optarg);
        break;
    case 'd':
        dst_dp = optarg;
        break;
    case 'h':
        dst_height = atoi(optarg);
        break;
    case 'f':
        src_fp = optarg;
        break;
    case 'o':
        dst_fp = optarg;
        break;
    case 'q':
        dst_quality = atoi(optarg);
        break;
    case 's':
        src_dp = optarg;
        break;
    case 'w':
        dst_width = atoi(optarg);
        break;
    default:
        usage();
        return 1;
        break;
}
}

```

The switch case and getopt bits should look familiar - not unlike the mmw program. There are some decisions that have to be made and instead of a lot of text - rely upon solid commenting to explain:

```

/* Run through a battery of checks before calling the main updater */
/* if this is a single image file - just pre-check now and handle */
/* determine the type then call the right function to scale it */
if ((src_fp != NULL) && (dst_fp != NULL)) {

    type = gettype(src_fp); /* call gettype to get the image type */

    if (strcmp(type, "jpeg") == 0) { /* if jpeg - use epeg */

```

```

        scale_peg(src_fp, dst_fp,
                 dst_height, dst_quality, dst_width);
    } else { /* if not jpeg fall through to imlib */
        scale_imlib2(src_fp, dst_fp, dst_height, dst_quality, dst_width);
    }

    return 0;
}

/* It is a directory not just an image - make sure the dir exists! */
if (check_handle(src_dp) != 0) {
    fprintf(stderr, "No input directory specified\n");
    usage();
    return 1;
}

/* Now we have to make sure that the dest dir is there and if not
go ahead and create it */
if (check_handle(dst_dp) != 0)
    if (mkdir(dst_dp, 0755)) {
        fprintf(stderr,
                "Could not create directory %s\n", dst_dp);
        return 1;
    }
}

```

One of the obvious things an astute reader may have picked up by now is the daemonize option. In the next section of code - if there is an interval defined: setup and execute a fork which calls the image rendering functions - otherwise call the same rendering functions once:

```

if (interval) {
    pid_t pid, sid;

    pid = fork();

    if (pid < 0) {
        exit (EXIT_FAILURE);
    } else if (pid > 0) {
        exit (EXIT_SUCCESS);
    }

    umask (0);

    sid = setsid();

    if (sid < 0)
        exit (EXIT_FAILURE);

    if ((chdir("/")) < 0)
        exit (EXIT_FAILURE);

    while (1) {
        update_image(src_dp, dst_dp, dst_height, dst_quality, dst_width);
        update_rescaled(src_dp, dst_dp);
        sleep(interval);
    }

    exit(EXIT_SUCCESS);
} else {
    update_image(src_dp, dst_dp, dst_height, dst_quality, dst_width);
    update_rescaled(src_dp, dst_dp);
}

```

```

    return 0;
}

```

In the preceding code notice the `setuid()` call; in the Perl section there was also a `setuid` call. When daemonizing `setuid()` is very much a constant but note how when daemonizing it is used but in the brute force forks program mentioned earlier it is not used on purpose. This is a difference of function and is important to notice in any program. The purpose of `etu` is to provide a clean quick method to rescale images (usually down) but the forks program is deliberately meant to fork and slam the Operating System as quickly and - if need be - messy as possible. As a programmer you are beholden to one rule and that is the program should do what it was meant to do as best as you can make it. Do not let dogma stand in the way of function - ever. In the case of `etu` being very safe and deliberate is a good thing ... conversely in the case of forks the opposite is true.

If only it were simple - there is still a lot to do. The `update_image()` function is the beginning of the end of our exhaustive program. it decides how to deal with each file in the associated directories:

```

void update_image(char *images, char *dst, int height, int quality, int width)
{
    int i;
    char dst_path[PATH_MAX]; /* The destination dir */
    char src_path[PATH_MAX]; /* The source dir */
    char * type;             /* What type of image? */
    struct dirent *src_dp;   /* Directory */
    DIR * src_dp_handle;    /* .. and the handle */

    src_dp_handle = opendir(images); /* Open the directory */

    i = 0; /* init our counter */

    /* Open up the directory and have at it ....*/
    while (src_dp = (readdir(src_dp_handle))) {
        if (i > -1) {
            strncpy(dst_path, fullpath(src_dp->d_name, dst),
                sizeof(src_dp->d_name) + 1);

                /* make sure everything is okay and setup the destination */
            if (check_handle(dst_path) != 0) {
                strncpy(src_path,
                    fullpath(src_dp->d_name, images),
                    sizeof(src_dp->d_name) + 1);

                    /* For each image get the type then call the right
                    function */
                type = gettype(src_path);
                if (strcmp(type, "jpeg") == 0) {
                    scale_peg(src_path, dst_path,
                        height, quality, width);
                } else {
                    scale_imlib2(src_path, dst_path, height,
                        quality, width);
                }
            }
        }

        i++; /* next one please */
    }

    closedir(src_dp_handle);
}

```

In a sense that function is the core algorithm and the last functions are more utility functions, they finally execute the decisions of the directory looper. The next function updates images:

```
void update_rescaled(char *images, char *dst)
{
    int i;
    char dst_path[PATH_MAX];
    char src_path[PATH_MAX];
    struct dirent *dst_img;
    DIR * dst_img_handle;

    dst_img_handle = opendir(dst);
    i = 0;
    while (dst_img = (readdir(dst_img_handle))) {
        if (i > -1) {
            strncpy(src_path, fullpath(dst_img->d_name, images),
                sizeof(dst_img->d_name) + 1);

            if (check_handle(src_path) != 0) {
                strncpy(dst_path, fullpath(dst_img->d_name, dst),
                    sizeof(dst_img->d_name) + 1);

                unlink(dst_path);
            }
        }
        i++;
    }
    closedir(dst_img_handle);
}
```

Now on to the last two functions, one function scales and imlib2 image and the next a jpeg only:

```
void scale_jpeg(char *jpeg, char *dstimg, int height, int quality, int width)
{
    Epeg_Image * jpeg_image;

    jpeg_image = epeg_file_open(jpeg);

    if (!jpeg_image) {
        fprintf(stderr, "Cannot open %s\n", jpeg);
        exit (1);
    }

    epeg_decode_size_set(jpeg_image, width, height);
    epeg_quality_set(jpeg_image, quality);
    epeg_file_output_set(jpeg_image, dstimg);
    epeg_encode(jpeg_image);
    epeg_close(jpeg_image);
}

void scale_imlib2(char *src, char *dst, int height, int quality, int width)
{
    Imlib_Image in_img, out_img;

    in_img = imlib_load_image(src);
    if (!in_img) {
        fprintf(stderr, "Unable to load %s\n", src);
        exit(1);
    }

    imlib_context_set_image(in_img);
```

```
out_img = imlib_create_cropped_scaled_image(0,0, imlib_image_get_width(),
                                             imlib_image_get_height(),
                                             width, height);

if (!out_img) {
    fprintf(stderr, "Failed to create scaled image\n");
    exit(1);
}

imlib_context_set_image(out_img);
imlib_save_image(dst);
}
```

... and we are done. Be sure to check the indexes for full program listings.

2.2.5 C Program: Network Decoder - ndecode

The last C program in this section is a packet decoder. Unlike the previous program involving network data this particular program looks exclusively at the payload of a network packet instead of the header. The ndecode program is actually relatively simple considering what it does. Ndecode achieves simplicity by leveraging the pcap library.

First a look at the top of the file, there are a lot of included files in this example and there are some compiler directives that say to include certain directories if we are on the NetBSD system:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/time.h>
#include <netinet/in_system.h>
#include <pcap.h>
#ifdef NETBSD
#include <net/ethernet.h>
#endif
#include <netinet/in.h>
#include <netinet/ip.h>
#include <signal.h>
#include <math.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netdb.h>
#include <semaphore.h>
#include <fcntl.h>
#include <getopt.h>
#include <errno.h>
#include <netinet/udp.h>
#include <net/if.h>
#ifdef NETBSD
#include <net/if_ether.h>
#endif
#include <sys/ioctl.h>
#include <time.h>
```

The reason for the directives is some of the structures used are not located in the same location on NetBSD as other systems.

Next is the PACKAGE macro which is just the name of the program:

```
#define PACKAGE "ndecode"
```

And now a boundary, note this is not a hard set boundary, but if the user does not specify the program defaults to capturing a maximum of 2048 packets then exiting:

```
#define MAXBYTES2CAPTURE 2048
```

Now how exactly should the program be controlled? That is to ask what options and arguments should be programmed? So far it is known that there should be an option for the number of polls. The program will be contextual to the interface where packets are read so an option to use an interface other than the first one

(which is how libpcap automatically figures out which one to use is done) needs to be added. Finally if there will be options then there should also be a usage print:

```

/* simple usage message */
static void usage()
{
    printf(PACKAGE " [option][arguments]\n"
           PACKAGE
           " "
           "[-i <interface>][-p <number>][-u]\n"
           "Options:\n"
           "-i <dev>   Specify the interface to watch\n"
           "-p <int>   Exit after analyzing int polls\n"
           "-u           Display help\n");
}

```

Note that since pcap is being used not unlike the previous example this program can take filter arguments as well. A function to copy off the filter is needed again:

```

/*
 * copy_argv: Copy off an argument vector
 *             except it does a lot of printing.
 * requires: argvector
 */
static char *copy_argv(char **argv)
{
    char **p;
    u_int len = 0;
    char *buf;
    char *src, *dst;
    p = argv;
    if (*p == 0)
        return 0;
    while (*p)
        len += strlen(*p++) + 1;
    buf = (char *)malloc(len);
    if (buf == NULL) {
        fprintf(stdout, "copy_argv: malloc");
        exit(EXIT_FAILURE);
    }
    p = argv;
    dst = buf;
    while ((src = *p++) != NULL) {
        while ((*dst++ = *src++) != '\0') ;
        dst[-1] = ' ';
    }
    dst[-1] = '\0';
    return buf;
}

```

Before moving onto the meat it is time to setup the main part of the program where arguments are parsed, assigned and the filter is set up. This should look familiar:

```

int main(int argc, char *argv[])
{
    struct bpf_program program;
    int i = 0;
    pcap_t *descr = NULL;
    char errbuf[PCAP_ERRBUF_SIZE];
    char *filter = NULL;
}

```

```

char *pcap_dev = NULL;
int c;
bpfuint32 mask;
bpfuint32 net;
uint32_t npolls = -1;
while ((c = getopt(argc, argv, "i:p:u")) != -1) {
    switch (c) {
        case 'i':
            pcap_dev = optarg;
            break;
        case 'p':
            if (optarg != NULL && isdigit(*optarg)) {
                npolls = atol(optarg);
                if (npolls < 0) {
                    fprintf(stderr,
                        "Packets must be > than 0\n");
                    return EXIT_FAILURE;
                }
            } else {
                fprintf(stderr, "Invalid packet number\n");
                return EXIT_FAILURE;
            }
            break;
        case 'u':
            usage();
            return EXIT_SUCCESS;
            break;
        default:
            usage();
            return EXIT_FAILURE;
            break;
    }
}
/* Got root? */
if (getuid()) {
    fprintf(stderr, "Must be root user.\n");
    return EXIT_FAILURE;
}
memset(errbuf, 0, PCAP_ERRBUF_SIZE);
/* Strip off any none getopt arguments for pcap filter */
if (!filter)
    filter = copy_argv(&argv[optind]);
/* Initialize the interface to listen on */
if (!pcap_dev)
    && ((pcap_dev = pcap_lookupdev(errbuf)) == NULL)) {
    fprintf(stderr, "%s\n", errbuf);
    return EXIT_FAILURE;
}
if ((descr = pcap_open_live(pcap_dev, 68, 0, 0, errbuf)) == NULL) {
    fprintf(stderr, "%s\n", errbuf);
    return EXIT_FAILURE;
}
pcap_lookupnet(pcap_dev, &net, &mask, errbuf); /* Get netinfo */
if (filter) {
    if (pcap_compile(descr, &program, filter, 0, net) == -1) {
        fprintf(stderr, "Error - `pcap_compile()'\n");
        return EXIT_FAILURE;
    }
}

if (pcap_setfilter(descr, &program) == -1) {
    fprintf(stderr, "Error - `pcap_setfilter()'\n");
    return EXIT_FAILURE;
}

```

```

    }

    pcap_freecode(&program);
}
pcap_loop(descr, npolls, payload_print, NULL);
/* Exit program */
printf("Closing capturing engine...\n");
pcap_close(descr);
return 0;
}

```

Note the main looks very similar to the other program that uses pcap with one exception, in the pcap loop it calls payload_print. Now onto the real meat, the part of the program that actually decodes and prints the packet data:

```

/*
 * Call libpcap and decode payload data.
 */
static void payload_print(u_char * arg, const struct pcap_pkthdr *header,
    const u_char * packet)
{
    int i = 0, *counter = (int *)arg;
    printf("Packet RECV Size: %d Payload:\n", header->len);
    for (i = 0; i < header->len; i++) {
        if (isprint(packet[i]))
            printf("%c ", packet[i]);
        else
            printf(". ");
        if ((i % 16 == 0 && i != 0) || i == header->len - 1)
            printf("\n");
    }
    return;
}

```

Doesn't look too difficult, here is some sample output:

```

# sudo ./ndecode -p 2
Packet RECV Size: 60 Payload:
. . 3 . . . . . E . .
. . . @ . @ . . . . .
. . J . F b . . . . ~ l . P . .
. . . . . * . . . . .
Packet RECV Size: 60 Payload:
. . . . . 3 . . . . . E . .
( . . @ . k . . . . .
. . F . J . ~ l . b . . . P . @
. u . . . . .
Closing capturing engine...

```

And yes, that data will print plain text passwords!

A.0.0 Miscellaneous Topics

The scope of this book is to look at as many compact examples of low level system programming in multiple languages in a fashion that is both usable and simple. In addition to the introductory material and examples there are some things that can be examined outside of the context of the book in appendices - just some side notes of interest I have collected and don't really fit within the scope of the core material.

A.0.1 Options Parsing

Options parsing can be difficult at times; to say the least. There exist a number of common methods and libraries to assist with options parsing. In this text, a look at writing option and argument parsing homespun and with a little help. Simple Parsing in sh

Simple parsing is easy in the shell:

```
while [ "$#" -gt "0" ]
do
    case $1 in
        -F)
            F_FLAG=1
            ;;
        -f)
            shift
            FILE_ARGUMENET=$1
            ;;
        -u)
            Usage
            exit 0
            ;;
        *)
            echo "Syntax Error"
            Usage
            exit 1
            ;;
    esac
    shift
done
```

Above, the input string is iterated over and particular options act or assign a variable. The posix getopt capability allows for built in - parsing:

```
while getopt ":f:Fu" opt; do
    case $opt in
        F) F_FLAG=1;;
        f) FILE_ARGUMENT=$OPTARG;;
        u) usage;;
        *) usage
            exit 1
            ;;
    esac
    shift
done
```

A colon after an option indicates it requires an argument. The getopt code is far more compact than the first example. What if the script requires long options? One approach is simply to hard code long options:

```
while [ "$#" -gt "0" ]
do
    case $1 in
        -F|--setflag)
            F_FLAG=1
            ;;
        -f|--file)
            shift
            FILE_ARGUMENET=$1
    esac
done
```

```

        ;;
    -u|--usage)
        Usage
        exit 0
        ;;
    *)
        echo "Syntax Error"
        Usage
        exit 1
        ;;
esac
shift
done

```

Setting up long options appears to be simple, however, it can quickly get out of control using the method show above. Instead, writing code to handle long options that can either be sourced in or easily dropped into scripts makes far more sense. Grigoriy Strokin has a good script that can either be copied in or sourced and can be found on his website. Following is the same code from above using getoptex:

```

. getoptx.sh
while getoptex "F; f; u. setflag file usage." "$@"; do
    F) F_FLAG=1;;
    f) FILE_ARGUMENT=$OPTARG;;
    u) usage;;
    *) usage
        exit 1
        ;;
done

```

It is pretty obvious that the single character is mapped to the the long option past the first . and the full terminator is the second dot. Of course, there is an even easier method as long as a few rules are observed:

```

while [ "$#" -gt "0" ]
do
    opt="{1/--}"
    opt=$(echo "${opt}" | cut -c 1 2>/dev/null)
    case $opt in
        F) F_FLAG=1;;
        f) shift;FILE_ARGUMENT=$1;;
        u) usage;;
        *) usage; exit 1;;
    esac
    shift
done

```

The problem with the last method is the long options are not hard-coded, the first character of the alpha string is cut and used as an option. In other words, --help and --heck will do the same thing. The idea is harmless except no options can be mixed and matched. Generally speaking, not having a --help and --heck valid in the same script or program should be avoided if possible. Options in Perl

With no case statement built in, doing options parsing in Perl can be a little tricky. Using the same example from the shell code above a simple options parser might look like: [1]

```

while ( my $arg = shift @ARGV ) {
    if ( $arg eq '-F' ) {
        $F_FLAG = 1;
    } elsif ( $arg eq '-f' ) {
        $FILE_ARGUMENT = shift @ARGV;
    }
}

```

```

    } elsif ( $arg eq '-u' ) {
        usage();
    } else {
        usage();
        exit 1;
    }
}

```

Relative to the shell, Perl seems a bit heavy handed in the amount of work needed. In Perl the options for handling are almost limitless. Associative arrays, hashes, arrays or just plain scalars arranged a certain way could be used.

Of course, another great thing about Perl is how simplistic string operations are handled. Using a method similar to the last shell method above can simplify the code a great deal:

```

for (my $argc = 0; $argc <= @ARGV; $argc++) {
    $opt = $ARGV[$argc];
    $opt =~ s/--//; # Get rid of 2 dashes
    $opt =~ s/-//; # Get rid of 1 dash
    $opt = substr($opt,0,1); # cut the first char
    if ($opt eq 'F') {
        $F_FLAG=1;
    } elsif ($opt eq 'f') {
        $FILE_ARGUMENT=$ARGV[++$argc];
    } elsif ($opt eq 'u') {
        usage();
    } else {
        usage();
        exit 1;
    }
}

```

Of course, the same two problems from the shell-code which cuts out the first alphanumeric exists; no two long options can start with the same letter and there is no verification of long options. Not unlike the shell, a simple list can be used to verify that long options are valid, following is an example sub routine:

```

...
my @valid_optlongs=("setflag", "file", "usage");
my @valid_optshort=("F",      "f",      "u");
...
sub parseopt{
    my ($opt) = shift;

    $opt =~ s/--//; # Get rid of 2 dashes
    $opt =~ s/-//; # Get rid of 1 dash

    if (scalar($opt) > 1) {
        for ($i = 0; $i < @valid_optlongs; $i++) {
            if ($opt eq $valid_optlongs[$i]) {
                return $valid_optshort[$i];
            }
        }
    } else {
        return $opt;
    }
}

```

Essentially instead of just trimming out the first valid alphanumeric, if the option is a long option check it against the list of valid long options and return the matching single byte option the long option correlates to.

Ultimately, using the getopt module should be done if it is available, why reinvent the wheel? Here is an example of using the Getopt module:

use Getopt::Std;

getopt ('f:uF');

```
die "Usage: $0 [ -f filename -u ]\n"
    unless ( $opt_f or $opt_u );

if ($opt_f) {
    my $filename = shift @ARGV;
} elsif ($opt_u) {
    usage();
    exit 0;
}
```

Definitely shorter and compact.

The oldest high level programming language - not unlike Perl - has many different approaches a programmer can take without using libraries:

```
int main(argc, argv)
    int argc;
    char *argv[];
{
    if (argc < 2) {
        printf("usage: %s number-of-execs sbrk-size job-name\n",
            argv[0]);
        exit(1);
    }
    ....
int main (argc, argv) {
    for (c = 0; c <=argc; c++) {
        if (argc[c] == 'F') {
            F_FLAG=1
        }
    }
    ...
}
```

libc offers up two levels of built in options handling, one for single options and one for long options. Since the options handling routines are in modern implementations, the examples will use GNU's version.

```
...
#include <getopt.h>
...
int main (int argc, char **argv)
{
    int c;
    char * file;

    while ((c = getopt(argc, argv, "F:f:u:")) != -1) {
        switch (c) {
            case 'F':
                F_FLAG=1
                break;
            case 'f':
                file = optarg;
                break;
            case 'u':
                ...
        }
    }
}
```

```

        usage();
        return 0;
        break;
    default:
        usage();
        return 1;
        break;
    }
}

```

Far more succinct than what may have happened using the previous C examples which would have been pretty spaghetti'd. Long options are even more interesting. The GNU C library internally handles assignment of long options by using the single alpha as the key inside of a data structure:

```

...
#include <getopt.>
...
int main(int argc, char **argv)
    while (1)
        {
            static struct option long_options[] =
                {
                    {"setflag", no_argument,      0, 'F' },
                    {"file",    required_argument, 0, 'f' },
                    {"usage",   no_argument,      0, 'u' },
                    {0,0,0,0} /* This is a filler for -1 */
                };

            int option_index = 0;

            c = getopt_long (argc, argv, "F:f:u:", long_options, &option_index);

            if (c == -1) break;

            switch (c) {
            case 'F':
                F_FLAG=1;
                break;
            case 'f':
                file = optarg;
                break;
            case 'u':
                usage();
                return 0;
                break;
            default:
                usage();
                return 1;
                break;
            }
        }
}

```

Short, sweet and to the point.

A.0.2 Using CPAN

Although the C libraries are quite extensive depending upon the system they can be difficult to track down and maintain. Perl on the other hand has a great deal of non core libraries and modules that can easily be downloaded and installed from the Comprehensive Perl Archive Network or CPAN for short. A much simpler method, however, is to use the CPAN shell which is provided with a base Perl install.

The cpan shell keeps the preferences for connections in \$HOME/.cpan. In order to use it, the shell must be initialized. Following are some of the example questions that come up - the long explanation that the cpan shell prints out have been omitted:

```
You don't seem to have a user configuration (MyConfig.pm) yet.
Do you want to create a user configuration now? (Y/n) [yes]
Would you like me to configure as much as possible automatically? [yes] No
CPAN build and cache directory? [/home/jrf/.cpan]
Download target directory? [/home/jrf/.cpan/sources]
Directory where the build process takes place? [/home/jrf/.cpan/build]
Store and re-use state information about distributions between
CPAN.pm sessions? [yes]
<prefs_dir>
Directory where to store default options/environment/dialogs for
building modules that need some customization? [/root/.cpan/prefs]
<auto_commit>
Always commit changes to config variables to disk? [no] Yes
<build_cache>
Cache size for build directory (in MB)? [100]
<index_expire>
Let the index expire after how many days? [1]
<scan_cache>
Perform cache scanning (atstart or never)? [atstart]
<cache_metadata>
Cache metadata (yes/no)? [yes]
<use_sqlite>
Use CPAN::SQLite if available? (yes/no)? [no]
<prerequisites_policy>
Policy on building prerequisites (follow, ask or ignore)? [ask]
<build_requires_install_policy>
Policy on installing 'build_requires' modules (yes, no, ask/yes,
ask/no)? [ask/yes] no
<check_sigs>
Always try to check and verify signatures if a SIGNATURE file is in
the package and Module::Signature is installed (yes/no)? [no] yes
<test_report>
Email test reports if CPAN::Reporter is installed (yes/no)? [no]
...
```

Most of the questions can use the default, but do read them all to be sure. Of most interest is the selection of download servers:

ftp:, or http: -- that host a CPAN mirror.

```
(1) Africa
(2) Asia
(3) Central America
(4) Europe
(5) North America
(6) Oceania
(7) South America
Select your continent (or several nearby continents) [] 5
(1) Bahamas
(2) Canada
(3) Mexico
(4) United States
Select your country (or several nearby countries) [] 4
(1) ftp://cpan-du.viaverio.com/pub/CPAN/
(2) ftp://cpan-sj.viaverio.com/pub/CPAN/
(3) ftp://cpan.cs.utah.edu/pub/CPAN/
```

```

(4) ftp://cpan.erlbaum.net/CPAN/
(5) ftp://cpan.hexten.net/
(6) ftp://cpan.llarian.net/pub/CPAN/
(7) ftp://cpan.mirrors.tds.net/pub/CPAN
(8) ftp://cpan.netnitco.net/pub/mirrors/CPAN/
(9) ftp://cpan.pair.com/pub/CPAN/
(10) ftp://ftp-mirror.internap.com/pub/CPAN/
(11) ftp://ftp.ccs.neu.edu/net/mirrors/ftp.funet.fi/pub/languages/perl/CPAN/
(12) ftp://ftp.dc.aleron.net/pub/CPAN/
(13) ftp://ftp.epix.net/pub/languages/perl/
(14) ftp://ftp.ncsu.edu/pub/mirror/CPAN/
(15) ftp://ftp.ndlug.nd.edu/pub/perl/
(16) ftp://ftp.osuosl.org/pub/CPAN/
34 more items, hit RETURN to show them
Select as many URLs as you like (by number),
put them on one line, separated by blanks, hyphenated ranges allowed

```

Generally I almost always pick the same ones I have used before:

```

Select as many URLs as you like (by number),
put them on one line, separated by blanks, hyphenated ranges allowed
e.g. '1 4 5' or '7 1-4 8' [] 3 9 10

```

A good litmus test is to update the CPAN bundle like so:

```

#
# install Bundle::CPAN
#

```

Depending upon which modules are already installed it may need to install dependancies. Note the syntax, the :: can also do multiple levels:

```

#
# install Net::FTP
#

```

When not doing a full bundle, the Bundle_Name::Module is the form. CPAN can also be searched from the shell:

```

#
# i/NAME/
#

```

The cpan shell can make a system programmer's life a lot easier.

A.0.3 The Basics of Make

The make utility is used to manage program build and installation. Make uses a Makefile which has directives stored in it. It also supports a wide variety of macros and other capabilities. This text will only examine the very basics of a Makefile.

In the simplest form a Makefile for the hello_world program might look like:

```
# Makefile
CC=/usr/bin/cc
all:
    ${CC} hello_world.c -o hello_world
```

Now type:

```
#
# make
#
```

Which defaults to the all target.

Make variables start at column 0 as do targets, however, directives that follow targets must be indented. The variables shown in the first example can also default to the user's PATH:

```
CC=gcc
MAKE=make
```

Which is generally the norm for handcrafted Makefiles. Note that there is nothing that cannot be stuffed into a variable in make. In the following example several source files are being combined into one resulting executable - all of which are defined by variables:

```
CC=gcc
SRCS= hello.c world.c io.c
BIN= hello_world
COPTS= -O2 --warn

hello: all

all:
    ${CC} ${COPTS} ${SRCS} -o ${BIN}
```

Library linking can also be specified:

```
# libs to add
LIBS= -lpcap
...
all:
    ${CC} ${COPTS} ${LIBS} ${SRCS} -o ${BIN}
```

Note that the use of brackets is simply the method I use as habit. GNU make generally uses parenthesis as well. Make has a great many other capabilities and there are several good online and dead tree resources for utilizing make.

A.0.4 Local Perl Libs

Sometimes a site has need for easily accessible Perl libraries. Three methods to go about keeping handy Perl code readily available are:

1. Just keep a code repository of functions and dump them into programs on an as needed basis.
2. Write a Perl library with a group of functions in them.
3. Write Perl modules.

A text on this site discusses the first option, this text deals with the second option. All three bear discussion though. Within this article, setting up a library (not module) will be addressed. Weighing In

Many factors have to be weighed to argue either for or against simply yank putting versus a local "requires" versus writing a module.

The yank put method seems appropriate for low frequency deployment. An example is a one time or limited time program. Many system administrators use this method because, well managing libraries is no fun. When a small set of programs or low frequency of development exists is the effort worth it?

Creating local libraries using the "requires" include method in Perl is a middle ground. A shared local set of libraries may be created that the Perl administrator does not have to solely maintain. A possible scenario is the line between sysadmin and system programmer. It is safe to say that the programmer is a competent programmer but may not know much about administering a Perl installation on the system. By creating a shared area on the filesystem, the sysadmin and programmer can deposit their libraries and indeed - benefit from each other's work.

Going the "full monty" of modules requires a bit more work. Modules can be setup with shared access. To keep downloaded modules from interfering with local modules, some administrative work needs to be done. If a module is good enough, it may be uploaded to CPAN and installed as part of the site install. Note that perl libs - just files with some functions in them - can also be uploaded to CPAN.

The last item to discuss is documentation. Method one has no real strict rules for documentation. Method two does not either, generally documentation is included. Method three requires documentation. Implementation Examples

Since yanking/putting functions from one file to another is pretty self explanatory, a look at creating a local library is presented. Implementation Details

At the imaginary site is an archive nfs server. On the server are directories packed with archives and a set of files listing archives. How the archives are organized is not important, what is important is the number of growing Perl programs and scripts that access them. These particular bits do the same operations over and over:

- Loading directories into arrays
- Loading files into arrays
- Reverse sorting
- Chomping every item in the directory list

The goal: put those tasks into a separate readable location to save time.

The Subroutines

```
##
# load_file: Open a file, read each line into an array, close the
#           file and return the array of lines.
##
sub load_file
{
    my ($file) = shift;
    my @file_contents;

    open(FILE, $file) or die "Unable to open file $file: $!\n";
    @file_contents = <FILE>;
    close FILE;

    return @file_contents;
}

##
# load_dir: Open a directory, read each entry into an array, close
#           the directory handle and return the array.
##
sub load_dir
{
    my ($dir) = shift;
    my @dir_contents;

    opendir(DIR, $dir) or die "Unable to open dir $dir: $!\n";
    @dir_contents = readdir(DIR);
    close(DIR);

    return @dir_contents;
}

##
# rsort: Sort then reverse an array.
##
sub rsort
{
    my (@contents) = @_;

    @contents = sort(@contents);

    return (reverse(@contents));
}

##
# atomic_chomp: Chomp every item on an array, return the array.
##
sub atomic_chomp
{
    my (@list2chomp) = @_;

    foreach(@list2chomp) {
        chomp($_);
    }

    return @list2chomp;
}
```

Documentation

Time to add documentation. Perldoc "tags" go in the source file as a stub. The markup is self explanatory:

```
__END__
=head1 DESCRIPTION
```

This small library provides four simple function: load a file into an array, load a directory into an array, sort and reverse an array and chomp every item in an array.

The usage for each is:

```
load_file(filename);
load_dir(dirname);
rsort(@array);
atomic_chomp(@array);
=head1 EXAMPLES
my @foo = load_dir("/home/mui");
@foo = rsort(@foo);
($item1, $item2) = atomic_chomp($item1, $item2);
=head1 PREREQUISITES
none
=head1 COREQUISITES
none
=cut
```

The now finished lib is ready:

```
##
# load_file: Open a file, read each line into an array, close the
#           file and return the array of lines.
##
sub load_file
{
    my ($file) = shift;
    my @file_contents;

    open(FILE, $file) or die "Unable to open file $file: $!\n";
    @file_contents = <FILE>;
    close FILE;

    return @file_contents;
}

##
# load_dir: Open a directory, read each entry into an array, close
#          the directory handle and return the array.
##
sub load_dir
{
    my ($dir) = shift;
    my @dir_contents;

    opendir(DIR, $dir) or die "Unable to open dir $dir: $!\n";
    @dir_contents = readdir(DIR);
    close(DIR);

    return @dir_contents;
}
```

```

##
# rsort: Sort then reverse an array.
##
sub rsort
{
    my (@contents) = @_;

    @contents = sort(@contents);

    return (reverse(@contents));
}

##
# atomic_chomp: Chomp every item on an array, return the array.
##
sub atomic_chomp
{
    my (@list2chomp) = @_;

    foreach(@list2chomp) {
        chomp($_);
    }

    return @list2chomp;
}

1; # Main - return a true value

__END__
=head1 DESCRIPTION

```

This small library provides four simple function: load a file into an array, load a directory into an array, sort and reverse an array and chomp every item in an array.

```

=head1 EXAMPLES
my @foo = load_dir("/home/mui");
@foo = rsort(@foo);
($item1, $item2) = atomic_chomp($item1, $item2);

=head1 PREREQUISITES

none

=head1 COREQUISITES

none

=cut

```

The only change is the addition of the ;1, a true return signal is needed to use the library. The Lib in Action

Time to take a look at the lib in action, first, the perldoc output:

```

[jrf@vela:~$] perldoc atomic.pl
ATOMIC(1)  User Contributed Perl Documentation ATOMIC(1)

DESCRIPTION
    This small library provides four simple function: load a file into an
    array, load a directory into an array, sort and reverse an array and

```

chomp every item in an array.

The usage for each is:

```
load_file(filename);
load_dir(dirname);
rsort (@array);
atomic_chomp (@array);
```

EXAMPLES

```
my @foo = load_dir("/home/mui");
@foo = rsort (@foo);
($item1, $item2) = atomic_chomp ($item1, $item2);
```

PREREQUISITES

none

COREQUISITES

none

perl v5.8.8

2006-05-16

ATOMIC (1)

An example of using the library:

```
#!/usr/bin/env perl
# atomtest - test out the atomic functions

use strict;

require "/path/to/atomic.pl";

my @dirlist = load_dir("/etc");
my @file    = load_file("/etc/passwd");

@dirlist = rsort (@dirlist);      # Sort and reverse the directory
@dirlist = atomic_chomp (@dirlist); # Chomp the directory list

listall (@dirlist);
listall (@file);
sub listall
{
    my @list2print = @_;

    foreach (@list2print) {
        print "$_\n";
    }
}
```

Instead of cluttering the output atomtest is piped to wc:

```
[jrf@vela:~$] ./atomtest | wc -l
244
```

Summary

Creating a Perl library and sharing via a path is easy. Creating any shared libraries in an organization can save time and reduce headaches.

A.0.5 Return Values

Return values, as proven by practically every piece of interface documentation written - do not have to be strings. Actually, a return value can in fact be whatever the programmer wants the return to be regardless of the language. This is important because some languages follow a return by sanity clause. The Perl language returns whatever the last operation results were - much like the shell. A program written in C/C++ or Java can do the same thing, the only difference is a few more hoops might need to be jumped through to implement a non string return on a function that was designed to return a string. On the flip-side, there is also the guaranteed null return. If a string operation fails for whatever reason, the program builds a real NULL string for the said platform (which should be handled by glibc or libc) and thus guarantees a proper NULL return. The last and perhaps most effective is just - set a string value that is an equivalent to a signal. Using an error string value is literally, done all of the time. A set of predefined string names within the local context of a program simply mean something bad. In short, `retval` does not mean just numbers.

Does this Make void and no-ops bad?

Nope, one of the big beefs out there is Perl does insist on returning something which - well just may not be wanted. If it can be void then why should a programmer have to say so? The reverse policy should be used instead, if an implicit return is desired, then do so, if not - do not force people to do so. One of the big upsides to the C language is how `retvals` can be tossed out the window. Usually, automatic returns do no harm. There are cases when input buffering is being parsed by sub routines in Perl and shell scripts that can make forced returns a problem, such as buffer mangling, but in general - harmless. Using void functions is completely legitimate since 9 times out of 10 they are inconsequential.

Case Study: Checker Shell Script

There exist so many shell scripts that check system status that if one had a penny for each they may not be rich but they would be well to do. One of the dangers of shell scripting is in-the-box thinking, especially when on a team. In this case study there are two examples of a very simple shell script that exhibits two different behaviors.

The first script is the bad one, it acts only for itself and does nothing to help anyone else:

```
...
for i in $#; do
    ssh $i uptime || logger -i -s "$0 could not reach ${i}"
done
...
exit 0
```

While `syslog` may be getting monitored somewhere, what if one wanted to use this "script" from another script?

A better script might offer a few alternatives to just logging:

```
...
sshchk()
{
    errors=0

    for i in $#; do
```

```

        ssh $i uptime
        if [ $? -gt 0 ]; then
            logger -i -s "SSH Connect error to ${i}"
            errors=$((errors+1))
        done

        return $errors
    }
    ...
    exit $errors

```

Now at least the caller - shell or script - will know there was a problem.

It may even be permissible to say "at least one check failed" and do the following:

```

...
sshchk()
{
    errors=0

    for i in $#; do
        ssh $i uptime
        if [ $? -gt 0 ]; then
            logger -i -s "SSH Connect error to ${i}"
            errors=$((errors+1))
        done

        if [ "$error" -gt 0 ]; then
            return 1
        fi
    }
    ...
}

```

In which case, something went wrong it just not known how many times.

In C Please

Many ideas in any programming language first come from inline testing. In line testing is a nice way of saying "cram it in using ifdefs." A good example of using returns in C is when if then else is not needed or may not even be applicable. Case Study: On the Side

In the example below, using pseudocode, a mount point is being checked and then an additional check is added, error on group or world readable. It does not matter if it is C, Perl or Java - the idea is the same. One version checks within the existing body of code while another, very succinctly, does not.

```

if MOUNTPPOINT
    return 0
endif

```

In Line Version

```

if MOUNTPPOINT
    if EXEC
        return 1
    if WORLDREAD
        return 1
    if GROUPREAD

```

```

        return 1
    return 0

```

Using retval to save the day...

```

    if MOUNTPPOINT
        retval=check_mount_perms
        return retval
    ...
    check_mount_perms
        if EXEC
            return 1
        if WORLDREAD
            return 1
        if GROUPREAD
            return 1
    return 0
    ...

```

So what does the additional function do? It does two things, one it takes the complexity of the check out of the simple mount point check. Next, it offers the ability to add or alter the checks on an as needed basis. What if checking for SGID or GGID were needed? What if just group and user were needed? In the latter version, such checks can be added without obfuscating the mount check too much.

What if more is needed? It is much easier to do this:

```

    if MOUNTPPOINT
        retval=check_mount_perms
        return retval
    ...
    check_mount_perms
        if EXEC
            return 1
        if WORLDREAD
            return 1
        if GROUPREAD
            return 1
        if GGID != MYGROUP
            return 1
    return 0
    ...

```

Versus adding yet another check in the main calling program.

How and When is too Much?

The question of how much message passing versus actually doing work is as old as computational devices. There is no simple answer. The best answer, and I yet again refer to Paul Graham - is do what seems natural. Just keep in mind that well formed returns, whether they are strings, numbers or NULL are ultimately up to the writer - not the machine. When not to be Prudent

As mentioned earlier there are times when a program does not need to return a value, so when might that be? One good example is just information, the classic usage:

```

    void usage(void) {
        printf("Here is some info...\n");
    }

```

}

always applies everywhere. A simple echo command etc. is just fine, even proven known operations, like flag checking, work great without needing explicit return values.

Summary

Return values help users and programmers alike every day. Making prudent use of them as a shell scripter, Perl monger or C programmer just makes it easier for all of us. The key part to remember is judicious use, if a check seems intrusive - just push it into a module and return something - otherwise just try to do the right thing.

A.0.6 Goofy Snippets

Over the years I have tried to keep a record of small interesting snippets I have found useful and somewhat out of the way. This appendice shows some of the more interesting ones.

A.0.6.i Inline Bash Function

All this stuffed into a bashrc:

```
TotalBytes=0
for Bytes in $(ls -l | grep "^-" | awk '{ print $5 }')
do
  let TotalBytes=$TotalBytes+$Bytes
done
TotalMeg=$(echo -e "scale=3 \n$TotalBytes/1048576 \nquit" | bc)
echo -n "$TotalMeg"
rand_titles ()
{
  if [ $TERM = xterm ]; then
    T_LINES=`wc -l $HOME/.xterm_titles | awk '{print $1;}'`
    T_LINE=$(( ${RANDOM} % $T_LINES );
    T_LINE=$(( T_LINE + 1 );
    TITLE=`head -$T_LINE < $HOME/.xterm_titles | tail -1`
    echo -ne "\033]0;${TITLE}\007"
  fi
}
PS1="[\u@\h:\w (\$(lsbytesum) Mb)]\$ "
PROMPT_COMMAND='rand_titles'
```

Results in taking random titles out of .xterm_titles and making them the xterminal title plus showing filesizes.

A.0.6.ii Tiny Inline ASM in C

Although it rare these days (excepting video games perhaps) that one ever needs to do so here is a tiny example of including assembler in C:

```
/* Simplistic example */
int
main(void)
{
  __asm__ (" mov %ah,5");
  return 0;
}
```

A.0.6.iii Tiny Inline C in Perl

Well if we looked at ASM in C it seems only fair to look at inline C in Perl:

```
#!/usr/bin/perl

use Inline C => <<'END_C';
void greet() {
    printf("Hello World\n");
}
```

```
END_C

greet;
```

A.0.6.iv ISBN Processing Using LISP

The List Processing Language of LISP is very interesting and worth getting to know at a basic level. Here is a snippet of code that does the math for ISBN processing. An ISBN is built using four sets of numbers:

1. group code
2. publisher code
3. id number assigned by publisher
4. check digit

Using the number of 0-673-38582-5.

The check digit has 11 possible values: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, X(10)

The final digit is determined by multiplying the first 9 digits by 10, 9, 8, 7, 6, 5, 4, 3, 2 respectively then add them (in our example we will call this y). The check digit is then chosen so that y+check_digit is divisible by 11.

```
[6]> (setq y-first (+ (* 10 0) (* 9 6) (* 8 7) (* 7 3) (* 6 3)))
149
[7]> (setq y-second (+ (* 5 8) (* 4 5) (* 3 8) (* 2 2)))
88
[8]> (setq y (+ y-first y-second))
237
[9]> (setq check-digit 5)
5
[10]> (setq mod 11)
11
[11]> (+ y check-digit)
242
[12]> (/ 242 mod)
22
```

A.0.6.v Elementary Congruence in Python

Elementary numerical congruence says m and n are integers and $m \neq 0$. Using the division algorithm n can be expressed as:

```
#
# n = qm + r, where 0 <= r < |m|
#
```

q is the quotient, r the remainder and |m| is absolute m. To prove this statement, a simple python session can help out:

```
>>>q = 3
>>>m = 9
>>>r = 7
>>>q * m + r
>>>34
```

A.0.6.vi NASM Boot Floppy

While trying to troubleshoot a BIOS problem, I found this and made some minor modifications to it.

; read from a floppy to boot up

[ORG 0]

jmp 07C0h:start ; Goto segment 07C0

```
; message string
msg      db      'Booting....'

start:
        ; Update the segment registers
        mov ax, cs
        mov ds, ax
        mov es, ax

        mov si, msg      ; put the message string into si

reset:
        ; Reset the floppy drive
        mov ax, 0
        mov dl, 0      ; Drive=0 (=A)
        int 13h
        jc reset      ; ERROR => reset again

read:
        mov ax, 1000h      ; ES:BX = 1000:0000
        mov es, ax
        mov bx, 0

        mov ah, 2      ; Load disk data to ES:BX
        mov al, 5      ; Load 5 sectors
        mov ch, 0      ; Cylinder=0
        mov cl, 2      ; Sector=2
        mov dh, 0      ; Head=0
        mov dl, 0      ; Drive=0
        int 13h
        ; Read!

        jc read      ; ERROR => Try again

        jmp 1000h:0000      ; Jump to the program

times 510-($-$$) db 0
dw 0AA55h
```