UNIX Administration Course

Copyright 1999 by Ian Mapleson BSc.

Version 1.0

mapleson@gamers.org
Tel: (+44) (0)1772 893297
Fax: (+44) (0)1772 892913
WWW: http://www.futuretech.vuurwerk.nl/

Detailed Notes for Day 2 (Part 4)

UNIX Fundamentals: Further Shell scripts.

for/do Loops.

The rebootlab script shown earlier could be rewritten using a for/do loop, a control structure which allows one to execute a series of commands many times.

Rewriting the rebootlab script using a for/do loop doesn't make much difference to the complexity of this particular script, but using more sophisticated shell code is worthwhile when one is dealing with a large number of systems. Other benefits arise too; a suitable summary is given at the end of this discussion.

The new version could be rewritten like this:

```
#!/bin/sh
for machine in akira ash cameron chan conan gibson indiana leon merlin \
               nikita ridley sevrin solo stanley warlock wolfen woo
do
    echo $machine
    rsh $machine init 6&
done
```

The '\' symbol is used to continue a line onto the next line. The 'echo' line displays a comment as each machine is dealt with.

This version is certainly shorter, but whether or not it's easier to use in terms of having to modify the list of host names is open to argument, as opposed to merely commenting out the relevant lines in the original version. Even so, if one happened to be writing a script that was fairly lengthy, eg. 20 commands to run on every system, then the above format is obviously much more efficient.

Similarly, the remountmapleson script could be rewritten as follows:

```
#!/bin/sh
for machine in yoda akira ash cameron chan conan gibson indiana leon merlin \
               nikita ridley sevrin solo stanley warlock wolfen woo
do
    echo $machine
    rsh $machine "umount /mapleson && mount /mapleson"
done
```

Note that in this particular case, the command to be executed must be enclosed within quotes in order for it to be correctly sent by rsh to the remote system. Quotes like this are normally not needed; it's only because rsh is being used in this example that quotes are required.

Also note that the '&' symbol is not used this time. This is because the rebootlab procedure is

asynchronous, whereas I want the remountdir script to output its messages just one action at a time.

In other words, for the rebootlab script, I don't care in what order the machines reboot, so each rsh call is executed as a background process on the remote system, thus the rebootlab script doesn't wait for each rsh call to return before progressing.

By contrast, the lack of a '&' symbol in remountdir's rsh command means the rsh call must finish before the script can continue. As a result, if an unexpected problem occurs, any error message will be easily noticed just by watching the output as it appears.

Sometimes a little forward thinking can be beneficial; suppose one might have reason to want to do exactly the same action on some other NFS-mounted area, eg. /home, or /var/mail, then the script could be modified to include the target directory as a single argument supplied on the command line. The new script looks like this:

```
#!/bin/sh
for machine in yoda akira ash cameron chan conan gibson indiana leon merlin \
               nikita ridley sevrin solo stanley warlock wolfen woo
do
    echo $machine
    rsh $machine "umount $1 && mount $1"
done
```

The script would probably be renamed to remountdir (whatever) and run with:

```
remountdir /mapleson
```

or perhaps:

```
remountdir /home
```

if/then/else constructs.

But wait a minute, couldn't one use the whole concept of arguments to solve the problem of communicating to the script exactly which hosts to deal with? Well, a rather useful feature of any program is that it will always return a result of some kind. Whatever the output actually is, a command always returns a result which is defined to be true or false in some way.

Consider the following command:

```
grep target database
```

If grep doesn't find 'target' in the file 'database', then no output is given. However, as a program that has been called, grep has also passed back a value of 'FALSE' - the fact that grep does this is simply invisible during normal usage of the command.

One can exploit this behaviour to create a much more elegant script for the remountdir command. Firstly, imagine that I as an admin keep a list of currently active hosts in a file called 'live' (in my case, I'd probably keep this file in /mapleson/Admin/Machines). So, at the present time, the file would contain the following:

```
yoda
akira
ash
```

```
cameron
chan
conan
gibson
indiana
leon
merlin
nikita
ridley
sevrin
solo
stanley
warlock
wolfen
woo
```

ie. the host called spock is not listed.

The remountdir script can now be rewritten using an if/then construct:

```
#!/bin/sh
for machine in yoda akira ash cameron chan conan gibson indiana leon merlin \
               spock nikita ridley sevrin solo stanley warlock wolfen woo
do
  echo Checking $machine...

  if grep $machine /mapleson/Admin/Machines/live; then
    echo Remounting $1 on $machine...
    rsh $machine "umount $1 && mount $1"
  fi
done
```

This time, the complete list of hosts is always used in the script, ie. once the script is rewritten, it doesn't need to be altered again. For each machine, the grep command searches the 'live' file for the target name; if it finds the name, then the result is some output to the screen from grep, but also a 'TRUE' condition, so the echo and rsh commands are executed. If grep doesn't find the target host name in the live file then that host is ignored.

The result is a much more elegant and powerful script. For example, suppose some generous agency decided to give the department a large amount of money for an extra 20 systems: the only changes required are to add the names of the new hosts to remountdir's initial list, and to add the names of any extra active hosts to the live file. Along similar lines, when spock finally is returned to the lab, its name would be added to the live file, causing remountdir to deal with it in the future.

Even better, each system could be setup so that, as long as it is active, the system tells the server every so often that all is well (a simple script could achieve this). The server brings the results together on a regular basis, constantly keeping the live file up-to-date. Of course, the server includes its own name in the live file. A typical interval would be to update the live file every minutes. If an extra program was written which used the contents of the live file to create some kind of visual display, then an admin would know in less than a minute when a system had gone down.

Naturally, commercial companies write professional packages which offer these kinds of services and more, with full GUI-based monitoring, but at least it is possible for an admin to create home-made scripts which would do the job just as well.

/dev/null.

There is still an annoying feature of the script though: if grep finds a target name in the live file, the

output from grep is visible on the screen which we don't really want to see. Plus, the umount command will return a message if /mapleson wasn't mounted anyway. These messages clutter up the main 'trace' messages.

To hide the messages, one of UNIX's special device files can be used. Amongst the various device files in the /dev directory, one particularly interesting file is called /dev/null. This device is known as a 'special' file; any data sent to the device is discarded, and the device always returns zero bytes. Conceptually, /dev/null can be regarded as an infinite sponge - anything sent to it is just ignored. Thus, for dealing with the unwanted grep output, one can simply redirect grep's output to /dev/null.

The vast majority of system script files use this technique, often many times even in a single script.

Note: descriptions of all the special device files /dev are given in Appendix C of the online book, "IRIX Admin: System Configuration and Operation".

Since grep returns nothing if a host name is not in the live file, a further enhancement is to include an 'else' clause as part of the if construct so that a separate message is given for hosts that are currently not active. Now the final version of the script looks like this:

```
#!/bin/sh
for machine in yoda akira ash cameron chan conan gibson indiana leon merlin \
               spock nikita ridley sevrin solo stanley warlock wolfen woo
do
  echo Checking $machine...

  if grep $machine /mapleson/Admin/Machines/live > /dev/null; then
    echo Remounting $1 on $machine...
    rsh $machine "umount $1 && mount $1"
  else
    echo $machine is not active.
  fi
done
```

Running the above script with 'remountdir /mapleson' gives the following output:

```
Checking yoda...
Remounting /mapleson on yoda...
Checking akira...
Remounting /mapleson on akira...
Checking ash...
Remounting /mapleson on ash...
Checking cameron...
Remounting /mapleson on cameron...
Checking chan...
Remounting /mapleson on chan...
Checking conan...
Remounting /mapleson on conan...
Checking gibson...
Remounting /mapleson on gibson...
Checking indiana...
Remounting /mapleson on indiana...
Checking leon...
Remounting /mapleson on leon...
Checking merlin...
Remounting /mapleson on merlin...
Checking spock...
spock is not active.
Checking nikita...
Remounting /mapleson on nikita...
Checking ridley...
Remounting /mapleson on ridley...
Checking sevrin...
```

```
Remounting /mapleson on sevrin...
Checking solo...
Remounting /mapleson on solo...
Checking stanley...
Remounting /mapleson on stanley...
Checking warlock...
Remounting /mapleson on warlock...
Checking wolfen...
Remounting /mapleson on wolfen...
Checking woo...
Remounting /mapleson on woo...
```

Notice the output from grep is not shown, and the different response given when the script deals with the host called spock.

Scripts such as this typically take around a minute or so to execute, depending on how quickly each host responds.

The rebootlab script can also be rewritten along similar lines to take advantage of the new 'live' file mechanism, but with an extra if/then structure to exclude yoda (the rebootlab script is only meant to reboot the lab machines, not the server). The extra if/then construct uses the 'test' command to compare the current target host name with the word 'yoda' - the rsh command is only executed if the names do not match; otherwise, a message is given stating that yoda has been excluded. Here is the new rebootlab script:

```
#!/bin/sh
for machine in yoda akira ash cameron chan conan gibson indiana leon merlin \
               spock nikita ridley sevrin solo stanley warlock wolfen woo
do
  echo Checking $machine...

  if grep $machine /mapleson/Admin/Machines/live > /dev/null; then
    if test $machine != yoda; then
      echo Rebooting $machine...
      rsh $machine init 6&
    else
      echo Yoda excluded.
    fi
  else
    echo $machine is not active.
  fi
done
```

Of course, an alternative way would be to simply exclude 'yoda' from the opening 'for' line. However, one might prefer to always use the same host name list in order to minimise the amount of customisation between scripts, ie. to create a new script just copy an existing one and modify the content after the for/do structure.

Notes:

- All standard shell commands and other system commands, programs, etc. can be used in shell scripts, eg. one could use 'cd' to change the current working directory between commands.

- An easy way to ensure that a particular command is used with the default or specifically desired behaviour is to reference the command using an absolute path description, eg. /bin/rm instead of just rm. This method is frequently found in system shell scripts. It also ensures that the scripts are not confused by any aliases which may be present in the executing shell.

- Instead of including a raw list of hosts in the script at beginning, one could use other commands such as grep, awk, sed, perl and cut to obtain relevant host names from the

/etc/hosts file, one at a time. There are many possibilities.

Typically, as an admin learns the existence of new commands, better ways of performing tasks are thought of. This is perhaps one reason why UNIX is such a well-understood OS: the process of improving on what has been done before has been going on for 30 years, largely because much of the way UNIX works can be examined by the user (system script files, configuration files, etc.) One can imagine the hive of activity at BTL and Berkeley in the early days, with suggestions for improvements, additions, etc. pouring in from enthusiastic testers and volunteers. Today, after so much evolution, most basic system scripts and other files are probably as good as they're going to be, so efforts now focus on other aspects such as system service improvements, new technology (eg. Internet developments, NSD), security enhancements, etc. Linux evolved in a very similar way.

I learned shell programming techniques mostly by looking at existing system scripts and reading the relevant manual pages. An admin's shell programming experience usually begins with simple sequential scripts that do not include if/then structures, for loops, etc. Later on, a desire to be more efficient gives one cause to learn new techniques, rewriting earlier work as better ideas are formed.

Simple scripts can be used to perform a wide variety of tasks, and one doesn't have to make them sophisticated or clever to get the job done - but with some insightful design, and a little knowledge of how the more useful aspects of UNIX work, one can create extremely flexible scripts that can include error checking, control constructs, progress messages, etc. written in a way which does not require them to be modified, ie. external ideas, such as system data files, can be used to control script behaviour; other programs and scripts can be used to extract information from other parts of the system, eg. standard configuration files.

A knowledge of the C programming language is clearly helpful in writing shell scripts since the syntax for shell programming is so similar. An excellent book for this is "C Programming in a UNIX Environment", by Judy Kay & Bob Kummerfeld (Addison Wesley Publishing, 1989. ISBN: 0 201 12912 4).

Other Useful Commands.

A command found in many of the numerous scripts used by any UNIX OS is 'test'; typically used to evaluate logical expressions within 'if' clauses, test can determine the existence of files, status of access permissions, type of file (eg. ordinary file, directory, symbolic link, pipe, etc.), whether or not a file is empty (zero size), compare strings and integers, and other possibilities. See the test man page for full details.

For example, the test command could be used to include an error check in the rebootlab script, to ascertain whether the live file is accessible:

```
#!/bin/sh
if test -r /mapleson/Admin/Machines/live; then
  for machine in yoda akira ash cameron chan conan gibson indiana leon merlin \
                 spock nikita ridley sevrin solo stanley warlock wolfen woo
  do
    echo Checking $machine...

    if grep $machine /mapleson/Admin/Machines/live > /dev/null; then
      if test $machine != yoda; then
        echo Rebooting $machine...
        rsh $machine init 6&
      else
        echo Yoda excluded.
```

```
      fi
    else
      echo $machine is not active.
    fi
  done
else
  echo Error: could not access live file, or file is not readable.
fi
```

NOTE: Given that 'test' is a system command...

```
% which test
/sbin/test
```

...any user who creates a program called test, or an admin who writes a script called test, will be unable to execute the file unless one of the following is done:

- Use a complete pathname for the file, eg. /home/students/cmpdw/test
- Insert './' before the file name
- Alter the path definition ($PATH) so that the current directory is searched before /sbin (dangerous! The root user should definitely not do this).

In my early days of learning C, I once worked on a C program whose source file I'd called simply test.c - it took me an hour to realise why nothing happened when I ran the program (obviously, I was actually running the system command 'test', which does nothing when given no arguments except return an invisible 'false' exit status).

Problem Question 1.

Write a script which will locate all .capture.mv.* directories under /home and remove them *safely*. You will not be expected to test this for real, but feel free to create 'mini' test directories if required by using mkdir.

Modify the script so that it searches a directory supplied as a single argument ($1).

Relevant commands: find, rm

Tips:

- Research the other possible options for rm which might be useful.
- Don't use your home directory to test out ideas. Use /tmp or /var/tmp.

Problem Question 2.

This is quite a complicated question. Don't feel you ought to be able to come up with an answer after just one hour.

I want to be able to keep an eye on the amount of free disk space on all the lab machines. How could this be done?

If a machine is running out of space, I want to be able to remove particular files which I know can

be erased without fear of adverse side effects, including:

- Unwanted user files left in /tmp and /var/tmp, ie. files such as movie files, image files, sound files, but in general any file that isn't owned by root.
- System crash report files left in /var/adm/crash, in the form of unix.K and vmcore.K.comp, where K is some digit.
- Unwanted old system log information in the file /var/adm/SYSLOG. Normally, the file is moved to oSYSLOG minus the last 10 or 20 lines, and a new empty SYSLOG created containing the aforementioned most recent 10 or 20 lines.

a. Write a script which will probe each system for information, showing disk space usage.

b. Modify the script (if necessary) so that it only reports data for the local system disk.

c. Add a means for saving the output to some sort of results file or files.

d. Add extra features to perform space-saving operations such as those described above.

Advanced:

e. Modify the script so that files not owned by root are only removed if the relevant user is not logged onto the target system.

Relevant commands: grep, df, find, rm, tail, cd, etc.

UNIX Fundamentals: Application Development Tools.

A wide variety of commands, programs, tools and applications exist for application development work on UNIX systems, just as for any system. Some come supplied with a UNIX OS as-standard, some are free or shareware, while others are commercial packages.

An admin who has to manage a system which offers these services needs to be aware of their existence because there are implications for system administration, especially with respect to installed software.

This section does not explain how to use these tools (even though an admin would probably find many of them useful for writing scripts, etc.) The focus here is on explaining what tools are available and may exist on a system, where they are usually located (or should be installed if an admin has to install non-standard tools), and how they might affect administration tasks and/or system policy.

There tend to be several types of software tools:

1. Software executed usually via command line and written using simple editors, eg. basic compilers such as cc, development systems such as the Sun JDK for Java.

   Libraries for application development, eg. OpenGL, X11, Motif, Digital Media Libraries - such library resources will include example source code and programs, eg. X11 Demo

Programs.

In both cases, online help documents are always included: man pages, online books, hints & tips, local web pages either in /usr/share or somewhere else such as /usr/local/html.

2. higher-level toolkits providing an easier way of programming with various libraries, eg. Open Inventor. These are often just extra library files somewhere in /usr/lib and so don't involve executables, though example programs may be supplied (eg. SceneViewer, gview, ivview). Any example programs may be in custom directories, eg. SceneViewer is in /usr/demos/Inventor, ie. users would have to add this directory to their path in order to be able to run the program. These kinds of details are in the release notes and online books. Other example programs may be in /usr/sbin (eg. ivview).

3. GUI-based application development systems for all manner of fields, eg. WorkShop Pro CASE tools for C, C++, Ada, etc., CosmoWorlds for VRML, CosmoCreate for HTML, CosmoCode for Java, RapidApp for rapid prototyping, etc. Executables are usually still accessible by default (eg. cvd appears to be in /usr/sbin) but the actual programs are normally stored in application-specific directories, eg. /usr/WorkShop, /usr/CosmoCode, etc. (/usr/sbin/cvd is a link to /usr/WorkShop/usr/sbin/cvd). Supplied online help documents are in the usual locations (/usr/share, etc.)

4. Shareware/Freeware programs, eg. GNU, Blender, XV, GIMP, XMorph, BMRT. Sometimes such software comes supplied in a form that means one can install it anywhere (eg. Blender) - it's up to the admin to decide where (/usr/local is the usual place). Other types of software installs automatically to a particular location, usually /usr/freeware or /usr/local (eg. GIMP). If the admin has to decide where to install the software, it's best to follow accepted conventions, ie. place such software in /usr/local (ie. executables in /usr/local/bin, libraries in /usr/local/lib, header files in /usr/local/include, help documents in /usr/local/docs or /usr/local/html, source code in /usr/local/src). In all cases, it's the admin's responsibility to inform users of any new software, how to use it, etc.

The key to managing these different types of tools is consistency; don't put one shareware program in /usr/local and then another in /usr/SomeCustomName. Users looking for online source code, help docs, etc. will become confused. It also complicates matters when one considers issues such as library and header file locations for compiling programs.

Plus, consistency eases other aspects of administration, eg. if one always uses /usr/local for 3rd-party software, then installing this software onto a system which doesn't yet have it is a simple matter of copying the entire contents of /usr/local to the target machine.

It's a good idea to talk to users (perhaps by email), ask for feedback on topics such as how easy it is to use 3rd-party software, are there further programs they'd like to have installed to make their work easier, etc. For example, a recent new audio standard is MPEG3 (MP3 for short); unknown to me until recently, there exists a freeware MP3 audio file player for SGIs. Unusually, the program is available off the Net in executable form as just a single program file. Once I realised that users were trying to play MP3 files, I discovered the existence of the MP3 player and installed it in /usr/local/bin as 'mpg123'.

My personal ethos is that users come first where issues of carrying out their tasks are concerned. Other areas such as security, etc. are the admin's responsibility though - such important matters should either be left to the admin or discussed to produce some statement of company policy,

probably via consulation with users, managers, etc. For everyday topics concerning users getting the most out of the system, it's wise for an admin to do what she/he can to make users' lives easier.

General Tools (editors).

Developers always use editing programs for their work, eg. xedit, jot, nedit, vi, emacs, etc. If one is aware that a particular editor is in use, then one should make sure that all appropriate components of the relevant software are properly installed (including any necessary patches and bug fixes), and interested users notified of any changes, newly installed items, etc.

For example, the jot editor is popular with many SGI programmers because it has some extra features for those programming in C, eg. an 'Electric C Mode'. However, a bug exists in jot which can cause file corruption if jot is used to access files from an NFS-mounted directory. Thus, if jot is being used, then one should install the appropriate patch file to correct the bug, namely Patch 2051 (patch CDs are supplied as part of any software support contract, but most patches can also be downloaded from SGI's ftp site).

Consider searching the vendor's web site for information about the program in question, as well as the relevant USENET newsgroups (eg. comp.sys.sgi.apps, comp.sys.sgi.bugs). It is always best to prevent problems by researching issues beforehand.

Whether or not an admin chooses to 'support' a particular editor is another matter; SGI has officially switched to recommending the nedit editor for users now, but many still prefer to use jot simply because of familiarity, eg. all these course notes have been typed using jot. However, an application may 'depend' on minor programs like jot for particular functions. Thus, one may have to install programs such as jot anyway in order to support some other application (dependency).

An example in the case of the Ve24 network is the emacs editing system: I have chosen not to support emacs because there isn't enough spare disk space available to install emacs on the Indys which only have 549MB disks. Plus, the emacs editor is not a vendor-supplied product, so my position is that it poses too many software management issues to be worth using, ie. unknown bug status, file installation location issues, etc.

Locations: editors are always available by default; executables tend to be in /usr/sbin, so users need not worry about changing their path definition in order to use them.

All other supplied-as-standard system commands and programs come under the heading of general tools.

Compilers.

There are many different compilers which might have to be installed on a system, eg.:

```
Programming     Compiler
Language        Executable

C               cc, gcc
C++             CC
Ada             ?
Fortran77       f77
Fortran90       f90
```

```
Pascal          ?
```

Some UNIX vendors supply C and C++ compilers as standard, though licenses may be required. If there isn't a supplied compiler, but users need one, then an admin can install the GNU compilers which are free.

An admin must be aware that the *release versions* of software such as compilers is very important to the developers who use them (this actually applies to all types of software). Installing an update to a compiler might mean the libraries have fewer bugs, better features, new features, etc., but it could also mean that a user's programs no longer compile with the updated software. Thus, an admin should maintain a suitable relationship with any users who use compilers and other similar resources, ie. keep each other informed of relevant issues, changes being made or requested, etc.

Another possibility is to manage the system in such a way as to offer multiple versions of different software packages, whether that is a compiler suite such as C development kit, or a GUI-based application such as CosmoWorlds. Multiple versions of low-level tools (eg. cc and associated libraries, etc.) can be supported by using directories with different names, or NFS-mounting directories/disks containing software of different versions, and so on. There are many possibilities - which one to use depends on the size of the network, ease of management, etc.

Multiple versions of higher-level tools, usually GUI-based development environments though possibly ordinary programs like Netscape, can be managed by using 'wrapper' scripts: the admin sets an environment variable to determine which version of some software package is to be the default; when a system is booted, the script is executed and uses the environment variable to mount appropriate directories, execute any necessary initialisation scripts, background daemons, etc. Thus, when a user logs in, they can use exactly the same commands but find themselves using a different version of the software. Even better, an admin can customise the setup so that users themselves can decide what version they want to use; logging out and then logging back in again would then reset all necessary settings, path definitions, command aliases, etc.

MPC operates its network in this way. They use high-end professional film/video effects/animation tools such as Power Animator, Maya, Flame, etc. for their work, but the network actually has multiple versions of each software package available so that animators and artists can use the version they want, eg. for compatibility reasons, or personal preferences for older vs. newer features. MPC uses wrapper scripts of a type which require a system reboot to change software version availability, though the systems have been setup so that a user can initiate the reboot (I suspect the reboot method offers better reliability).

Locations:

Executables are normally in /usr/sbin, libraries in /usr/lib, header files in /usr/include and online documents, etc. in /usr/share. Note also that the release notes for such products contain valuable information for administrators (setup advice) and users alike.

Debuggers.

Debugging programs are usually part of a compilation system, so everything stated above for compilers applies to debuggers as well. However, it's perfectly possible for a user to use a debugger that's part of a high-level GUI-based application development toolkit to debug programs that are created using low-level tools such as jot and xedit. A typical example on the Ve24 machines is students using the cvd program (from the WorkShop Pro CASE Tools package) to debug their C

programs, even though they don't use anything else from the comprehensive suite of CASE tools (source code management, version control, documentation management, rapid prototyping, etc.)

Thus, an admin must again be aware that users may be using features of high-level tools for specific tasks even though most work is done with low-level tools. Hence, issues concerning software updates arise, eg. changing software versions without user consulation could cause problems for existing code.

High-level GUI-based Development Toolkits.

Usually vendor-supplied or commercial in nature, these toolkits include products such as CosmoCode (Java development with GUI tools), RapidApp, etc. As stated above, there are issues with respect to not carrying out updates without proper consideration to how the changes may affect users who use the products, but the ramifications are usually much less serious than low-level programs or shareware/freeware. This is because the software supplier will deliberately develop new versions in such a way as to maximise compatibility with older versions.

High-level toolkits sometimes rely on low-level toolkits (eg. CosmoCode depends on the Sun JDK software), so an admin should also be aware that installing updates to low-level toolkits may have implications for their higher-level counterparts.

High-level APIs (Application Programming Interfaces).

This refers to advanced library toolkits such as Open Inventor, ViewKit, etc. The actual application developments tools used with these types of products are the same, whether low-level or high-level (eg. cc and commands vs. WorkShop Pro CASE Tools). Thus, high-level APIs are not executable programs in their own right; they are a suite of easier-to-use libraries, header files, etc. which users can use to create applications designed at a higher level of abstraction. Some example high-level APIs and their low-level counterparts include:

```
Lower-level          Higher-level

OpenGL               Open Inventor
X11/Motif            ViewKit/Tk
ImageVision          Image Format Library,
                     Electronic Light Table.
```

This is not a complete list. And there may be more than one level of abstraction, eg. Open Inventor is a subset of VRML.

Locations: high-level APIs tend to have their files stored in correspondingly named directories in /usr/lib, /usr/include, etc. For example, Open Inventor files can be found in /usr/lib/Inventor and /usr/include/Inventor. An exception is support files such as example models, images, textures, etc. which will always be in /usr/share, but not necessarily in specifically named locations, eg. the example 3D Inventor models are in /usr/share/data/models.

Shareware and Freeware Software.

This category of software, eg. the GNU compiler system, is usually installed either in /usr/local somewhere, or in /usr/freeware. Many shareware/freeware program don't have to be installed in one

of these two places (Blender is one such example) but it is best to do so in order to maintain a consistent software management policy.

Since /usr/local and /usr/freeware are not normally referenced by the standard path definition, an admin must ensure that relevant users are informed of any changes they may have to make in order to access newly installed software. A typical notification might be a recommendation of a how a user can modify her/his own .cshrc file so that shells and other programs know where any new executable files, libraries, online documents, etc. are stored.

Note that, assuming the presence of Internet access, users can easily download freeware/shareware on their own and install it in their own directory so that it runs from their home account area, or they could even install software in globally writeable places such as /var/tmp. If this happens, it's common for an admin to become annoyed, but the user has every right to install software in their own account area (unless it's against company policy, etc.) A better response is to appreciate the user's need for the software and offer to install it properly so that everyone can use it, unless some other factor is more important.

Unlike vendor-supplied or commercial applications, newer versions of shareware and freeware programs can often be radically different from older versions. GIMP is a good example of this - one version introduced so many changes that it was barely comparable to an older version. Users who utilise these types of packages might be annoyed if an update is made without consulting them because:

- it's highly likely their entire working environment may be different in the new version,
- features of the old version may no longer be available,
- aspects of the new version may be incompatible with the old version,
- etc.

Thus, shareware/freeware programs are a good example of where it might be better for admins to offer more than one version of a software package, eg. all the files for Blender V1.57 are stored in /usr/local/blender1.57_SGI_6.2_iris on akira and sevrin. When the next version comes out (eg. V1.6), the files will be in /usr/local/blender1.6_SGI_6.2_iris - ie. users can still use the old version if they wish.

Because shareware/free programs tend to be supplied as distinct modules, it's often easier to support multiple versions of such software compared to vendor-supplied or commercial packages.

Comments on Software Updates, Version Issues, etc.

Modern UNIX systems usually employ software installation techniques which operate in such way so as to show any incompatibilities before installation (SGIs certainly operate this way); the inst program (and thus swmgr too since swmgr is just a GUI interface to inst) will not allow one to install software if there are conflicts present concerning software dependency and compatibility. This feature of inst (and swmgr) to monitor software installation issues applies only to software subsystems that can be installed and removed using inst/swmgr, ie. those said to be in 'inst' format. Thankfully, large numbers of freeware programs (eg. GIMP) are supplied in this format and so they can be managed correctly. Shareware/Freeware programs do not normally offer any means by

which one can detect possible problems before installation or removal, unless the authors have been kind enough to supply some type of analysis script or program.

Of course, there is nothing to stop an admin using low-level commands such as cp, tar, mv, etc. to manually install problematic files by copying them from a CD, or another system, but to do so is highly unwise as it would invalidate the inst database structure which normally acts as a highly accurate and reliable record of currently installed software. If an admin must make custom changes, an up-to-date record of these changes should be maintained.

To observe inst/swmgr in action, either enter 'inst' or 'swmgr' at the command prompt (or select 'Software Manager' from the Toolchest which runs swmgr). swmgr is the easier to understand because of its intuitive interface.

Assuming the use of swmgr, once the application window has appeared, click on 'Manage Installed Software'. swmgr loads the inst database information, reading the installation history, checking subsystem sizes, calculating dependencies, etc. The inst system is a very effective and reliable way of managing software.

Most if not all modern UNIX systems will employ a software installation and management system such as inst, or a GUI-based equivalent.

Summary.

As an administrator, one should not need to know how to use the software products which users have access to (though it helps in terms of being able to answer simple questions), but one should:

- be aware of where the relevant files are located,
- understand issues concerning revision control,
- notify users of any steps they must take in order to access new software or features,
- aid users in being able to use the products efficiently (eg. using /tmp or /var/tmp for working temporarily with large files or complex tasks),
- have a consistent strategy for managing software products.

These issues become increasingly important as systems become more complex, eg. multiple vendor platforms, hundreds of systems connected across multiple departments, etc. One solution for companies with multiple systems and more than one admin is to create a system administration committee whose responsibilities could include coordinating site policies, dealing with security problems, sharing information, etc.