UNIX System Manager Manual (SMM)

Addendum

Integrated Solutions
1140 Ringwood Court
San Jose, CA 95131
(408) 943-1902

490198 Rev. A

December 1987

PREFACE

This document consists of training modules for the following commonly used system administration tasks:

- Automating System Processes
- Adding a User
- System Startup and Shutdown
- Using the Find Command

If you have not already done so, please take a few minutes to complete the two surveys in the introductory section.

Automating System Processes

Introduction

In running a UNIX computer system, one frequently encounters tasks that need to be accomplished after some delay and possibly repeated on a regular basis. These include resource intensive jobs, backups, cleaning up directories, sending reminder letters, polling UUCP connections, etc. The **at** and **cron** mechanisms of UNIX enable you to schedule such tasks. The **at** command can be used to submit a job to be executed at a later time, while regularly scheduled events can be automated with the **cron** utility.

Prerequisites

Before beginning this module you should be able to edit a file with vi.

Objective

After completing this module, you will be able to:

- use the **at** facility to execute commands after a specified delay; and
- use the **cron** facility to automate commands.

Procedures

Objective A
Using the at Mechanism to Send Phantom Mail

After all your hard work you are about to go on a well-earned vacation. The thought of being a whole week away from your work and your colleagues is of course a cause of concern to you: will they miss you? Will they remember you with the fondness you deserve? Well, fear not, you can send them a reminder of your existence while you are away, and let them figure it out.

You can do this by using the **at** utility.

A1 First create a file called, for instance, phantom:

vi *phantom*

In it enter the message you would like your colleagues to receive while you are counting grains of sand on the beaches of Waikiki — just try not to overdo it; you do have to survive their greetings when you come back. Save the file and, from the command line, enter

at *time*
at> **mail** *everybody* **<** *phantom*
CONTROL-d

where *time* is the time at which you want the command to be executed. This can be, in its simplest form, a two digit sequence meaning an hour, or a four digit sequence meaning hour and minutes. It can also be two or four digits followed by **am** or **pm**, to specify time of the day. Note that if **am** or **pm** do not follow the digits, a 24-hour clock is assumed (e.g., *1700* is equivalent to *5 pm*). In addition to this, you can specify a day of the week (e.g., Fri or Friday) and/or a month and a date (for instance Jan 24).

The above does not exhaust all the possibilities. The following are examples of possible entries:

at 0815am Jan 14
at 08:15
at 17:05
at noon Friday
at midnight January 24

Let's go back to our example:

at *time*
at> **mail** *everybody* **<** *phantom*
CONTROL-d

We now know what *time* means. We use the word *everybody* to indicate that you can have an alias for the names of all your colleagues, or you can enter their login names one after the other.

A2 Another way to set this up would be to modify the *phantom* file so that instead of being a simple text file it becomes a script file that you ask **at** to execute for you at the time and date specified. For this, open the *phantom* file and at the top enter the following:

```
mail everybody << EOF
Here put the text of the message you want to send.
It can occupy as many lines as you want.
EOF
```

A3 Now make sure that the file is executable; enter

chmod 740 phantom

and call up **at** to run the file:

at 12:30 Jan 24
at> *phantom*
CONTROL-d

A4 All files queued up for execution by **at** are located in **/usr/spool/at** during execution.

Objective B

Using Crontab to Ditch Those *tmp* Files

Let's say you have the nasty habit of creating a lot of temporary files while using UNIX. You give these files names starting with "*tmp*" but you never get around to deleting them. Eventually they take up a lot of disk space, and they never get used again. You can use the **cron** mechanism to remove all of your "*tmp*" files every night while you sleep.

B1 First, create a shell script to perform the task. In this example, let's call our script *cleanup* and put it in our home directory. A script to remove all files whose names start with *tmp* from your home directory and all of its descendant directories might consist only of the following line:

find */work/staff/ours* **-name** *"tmp*"* **-exec rm -f "{}" \;**

B2 Call up the file */usr/lib/crontab* for editing.

B3 Add the appropriate **crontab** entry to */usr/lib/crontab*. The entry is of the following form:

mah hod dom moy dow command ...

The first five fields are separated (or delimited) by spaces and each contains an integer specifying a time or times as follows:

- **mah** -- **m**inutes **a**fter **h**our. The number of minutes past the hours the task should be performed.
- **hod** -- **h**our **o**f **d**ay. The hour of the day (in twenty-four hour format) the task should be performed.
- **dom** -- **d**ay **o**f **m**onth. The day of the month (1-31) the task is to be performed.
- **moy** -- **m**onth **o**f **y**ear. The month (1-12) the task is to be performed. Note that if an impossible date, such as February 30, is specified, the task will never be performed.
- **dow** -- **d**ay **o**f **w**eek. The day of the week (1-7, with 1 representing Monday) the task is to be performed.

Any of the above integers may be replaced by a comma-separated list of integers (or an asterisk (*) to represent all possible values). The task will be performed at any time which matches the listed entries. For example, if the **hod** field contains *3,4,5* and the **mah** field contains *30,50,* and all of the other fields contain an asterisk (*), the task will be performed every day at 3:30 AM, 3:50 AM, 4:30 AM, 4:50 AM, 5:30 AM and 5:50 AM.

If we want our task to be performed every day at 3:30 AM, the entry we should add to */usr/lib/crontab* is:

*30 3 * * * /bin/sh /work/accounts/ours/cleanup*

Remember that our hypothetical script is called *cleanup* and our hypothetical home directory is */work/accounts/ours*. Notice

also that we are invoking the Bourne shell (**/bin/sh**) to run the script. The reason for this is that the Bourne shell loads faster than the C shell (**/bin/csh**), and not because our script requires it — this particular script would run under either shell. There may be times when you will need to specify a shell because the script is written using a specific shell's syntax.

Objective C
Making a Change

Suppose you discover that you are creating these nasty files not just in your own directories but in public directories throughout the system. You want to remove any files *owned by you* whose names begin with the letters *tmp*.

C1 Replace the one line in the file *cleanup* in your home directory with the following:

```
find -user ours -name "tmp*" -exec /bin/rm -f "{}" \;
```

C2 Note that we made no changes whatsoever to the **crontab** file! This is a major rationale behind using scripts instead of inserting commands directly into the **crontab** file. Now if you wanted to make an additional change (like only removing *tmp* files that are over 24 hours old, to prevent clobbering files that are in use) you could do so without having to access the **crontab** file.

Objective D
How cron Works

The **cron** process, unlike many other processes in UNIX, never exits once it is initiated—it stays in the background and checks the */usr/lib/crontab* file once every minute. It is therefore started every time the system is started up and killed every time the system is shut down.

If you look at the table of processes running in your system at any time, by entering the command **ps axu**, you will see that **cron** is not associated with any terminal and is run by **root**.

The above implies two things:

- Error messages should be dealt with, and
- File- and command-pathnames should be given specifically.

If you look at our examples above you will see that indeed in all cases we provided full pathnames not only for the files in question (for instance, */work/accounts/ours*) but also for the commands themselves (as in **/bin/sh**). This not only makes the process execute slightly faster, but does not depend on whether the command in question is findable through **root**'s path.

What about the error messages? Some utilities have options that suppress the output of error messages. Such is the case with **rm**. The **-f** option to **rm**, for instance, prevents **rm** from asking the user whether certain file permissions should be overlooked when removing the file or not, forcing the removal no matter what the permissions are. This option should always be used when the **rm** command is invoked in the **crontab** file. Other utilities do not have such options; in this case, the error messages that conditions such as non-existent files would produce are lost. If you want to capture these messages when they occur, you have to redirect them to a file—they will not appear on a terminal, since there is no terminal associated with **cron**.

Error messages are generally output to standard error, so you have to redirect standard error—an imaginary entry in /usr/lib/crontab doing this would look like:

30 02 * * * /bin/sh *command* **2>** */tmp/errlog*

It could also look like this:

30 02 * * * /bin/sh *command* **2>** */tmp/errlog* **1>** *output_file*

In the first case the command will be executed every day at

2:30 in the morning, and the standard error of the command will be redirected to */tmp/errlog*. In the second case, standard error will be redirected to */tmp/errlog* and standard output to a file called *output_file*.

Objective E

Securing the crontab File

The file */usr/lib/crontab* is owned by **root**, and the **cron** program is also run by **root**. This means that in order to make sure that unwanted programs are not run on the system as if **root** were running them you have to make sure of a few things.

One is making sure that at all times the file */usr/lib/crontab* is not writable by anyone but the owner (that is, **root**).

The other is making sure that processes that do not have to be run by root are run by the owner of the program file. This is done by preceding the *crontab* command entry with an **su** command that will substitute user id from **root** to the owner of the program file. For instance, **uucp** programs can be run as:

time **su uucp** *command*

where you would substitute the appropriate entries for *time* and *command*.

Failing to secure the *crontab* file and the entries in it can result in extremely serious security breaches. You may even consider having an entry in */usr/lib/crontab* that periodically makes sure that the file's ownership and permissions are set properly, such as

22 * * * * su root /etc/chown root /usr/lib/crontab 2> */dev/console*
23 * * * * su root /bin/chmod 744 /usr/lib/crontab 2> */dev/console*

This will make sure, every hour of every day, that the ownership and permissions have not been modified accidentally. It does not constitute a total defense against unwanted intrusions on the system.

Notes

Notes

Adding a User

Introduction

One of the tasks that you will need to perform from time to time is adding a new user to the system. While adding a user is not difficult, it does requires some advance planning to be successful.

Prerequisites

Before beginning this module, you should be able to edit files using the editor; create and move among UNIX directories. You should also be able to read, understand, and modify the access permissions of UNIX files.

Objective

After completing this module you will be able to add and remove users and modify a user's working environment.

Objective A
Preparing to Add a User

A1 In this section you will learn the steps needed prior to adding a user to the UNIX computer. Before we get going, turn to the end of this module and take out the sheet titled *New User Information Sheet.* Throughout this section you will be adding entries to this sheet. While you may not understand what all the entries on this sheet mean, they will become clear as you work through the module. This sheet, or one like it of your own creation, should be used whenever you add a new user.

As a practice exercise, you will add a fictitious user to your system. Later in the module, you will remove the user so you won't have any extra personalities running around in the system.

A2 **The *letc/passwd* File:**

The first file we are going to look at is the *passwd* file, located in the */etc* directory. This file contains each user's password, along with other information that the system needs to set up a user's working environment. Look at this file using the following command:

more */etc/passwd*

Each line is a separate entry consisting of seven different fields, each field separated by a colon. This is what a typical entry in the */etc/passwd* file would look like:

tut:p6zbNENKMisjl:47:101:as in the king:/staff/tut:/bin/csh

While this file may look chaotic, each line is consistent in its pattern. The pattern is as follows:

login:password:uid:gid:miscellaneous:home directory:start up shell

A3 **login**: The *login* field is the name given to the person so that the computer knows who it is talking to. The only two real constraints in choosing a login name is that it has to be unique, so no two users have the same login name, and it should also be written in lower case letters.

On the New User Information Sheet find the line that says *Login name.* For our fictitious user we are going to turn to the world of entertainment, so write down the name *daffy* in the space.

Above, we said that one of the constraints for choosing a login name is that it had to be unique. To make sure there is nobody named *daffy* already on the system, enter the following line:

grep *daffy /etc/passwd*

If there is already a *daffy* on the system just substitute a different name for this module. Then try to find him and get his autograph. (Don't be too persistent, though, or you might ruffle his feathers.)

A4 **password**: The password entry is an encrypted copy of the person's password that only the computer can read. You may notice that on the New User Info' Sheet the space for *password* tells you to leave it blank. When you first enter a new user in the */etc/passwd* file this field should be left blank. Later in the module you will assign *daffy* a password using the **passwd** command.

A5 **uid**: The initials *uid* stand for **u***ser* **id**. Like the login name the user's *uid* must be unique. No two users should have the same *uid.*

The easiest way to assign a user a *uid* number is to find the largest number in use and add one to it. To find the largest

number in this field, enter the following command from the shell:

awk -F: '{print $3}' */etc/passwd* **| sort -n | tail -1**

In this command, the **awk** command looks at the file */etc/passwd* and separates the third field, *uid* numbers. These numbers are piped to the **sort** command which arranges them in reverse numerical order and pipes them to the **tail** command. The **tail** command then takes the last number and displays it on the screen.

To choose a *uid* number for the new user simply add 1 to the highest number displayed. For instance, if the output from this command is

36

the number for the new user should be **37**. When you have arrived at the correct number write it down on the appropriate line on the New Users Information Sheet.

A6 **gid**: The initials *gid* in the fourth field stand for g*roup* **id**. This is a number that establishes the user's default group when he or she logs in. Each user must be a member of at least one group.

To find out which groups are available enter the following command:

more */etc/group*

There are four fields in each entry, each field separated by a colon. A typical entry should look like the following:

guest::31:root,bob,carol,ted,alice

The fields are as follows:

name:password:number:members

The first field is the group's *name*. The second field is for the group *password* if it is needed. This field should be left blank in this version of UNIX. The third field is the *group ID* number and the last field is a list of *members* of the group.

The lists in the fourth field do not necessarily contain all the users in the group. If a user has a certain group as their default group then his or her name does not have to appear in the list of members. However, to be a member of a group other than the default group, each user must have his or her name in the respective group.

There may be any number of groups in the file. The group names are purely arbitrary. For our user it doesn't really matter which group you choose so just pick the one that looks like other staff users might belong to it. We don't want *daffy* to feel left out.

Now look at the third field in whichever group you chose and write the number in the appropriate space on the New User Info' Sheet.

A7 **miscellaneous**: This field is available to enter any miscellaneous information about the user such as their phone number or job title or any thing you may find useful. This field can also be left blank if you choose.

Note: We recommend leaving this field blank. You can then use the **chfn** command (described below) to add information to this field in a standard format.

A8 **home directory**: Each user should have their own home directory when they log in. This field establishes where the user's home directory will be located. It will also be where the computer looks for the user's *.login* or *.profile* file depending on which shell they are assigned. For convenience sake it is usually easiest to establish one directory that contains only user's home directories.

For now put *daffy*'s home directory in the same directory as your home directory and write the full path name in the New Users Info' Sheet. For example, if your home directory is */usr/you*, write down */usr/daffy*.

A9 **start up shell**: The final column tells the computer which shell the user will work from when he or she logs in. There are basically two shells available to users of the UNIX system: the Bourne shell and the C shell. The path names are **/bin/sh** and **/bin/csh** respectively. During this module we will be using the C shell but either shell may be used.

In the New Users Info' Sheet write the full path name of the C shell, */bin/csh*, or whichever shell you choose.

A10 By now your New User Info' Sheet should look something like the following:

Login name:_daffy_____

Password: (Leave this blank for a new user)

uid (user id number)__45_____

gid (group id number)_32_____

Miscellaneous:__(blank)_____

Users home directory (full path name)_/usr/daffy_____

Start up shell (full path name)__/bin/csh_____

Objective B
Actually Adding the User

B1 Now that you have all the information you need to add **daffy** to the system, it is time to actually do it. From this point on you will need to log in as root to make the following changes. Start by making a copy of the */etc/passwd* file so that if you make a mistake you have a backup ready.

cp */etc/passwd /etc/passwd.old*

B2 Open the *passwd* file with **vi:**

vipw

The **vipw** stands for **vi**sually edit the **p**ass**w**ord file. This command makes a temporary copy of the password file and allows you to edit it. When you are done, the original */etc/passwd* file is replaced by the temporary file. The advantage of the **vipw** command is that it may not be used by two people at once. If two users edit the same file at the same time, confusion usually results.

Move to the end of the file and type **o** to open a new line. Enter the information from the New Users Sheet using the same structure as the rest of the file (seven fields, separated by colons, etc). The line should look something like this:

daffy::32:45::/usr/daffy:/bin/csh

For your entry, the fields for *uid*, *gid* and *home directory* may be different than the ones here.

Now close the file using **:wq**.

Objective C
Giving daffy a Home

C1 In the home directory section of the New User Info' Sheet you wrote down the full path name of the user's new home directory. As you know, directories do not spontaneously appear out of thin air. Using the following command, make a new home directory using the full path name as it appears in the New User Sheet.

mkdir */usr/daffy*

Obviously, you should replace */usr/daffy* with whatever *home directory* you specified on the New User Sheet.

Now **cd** to the directory above *daffy*'s home directory and enter:

ls -lgd *daffy*

The output from this command should look something like this:

drwxrwxrwx 1 root sys july 4 12:34 daffy

From the output you can see that, not only does *daffy* not own his home, but he's not even a member of the right group. But fear not, we will soon set his life in order.

C2 The first thing you will do is change the ownership of the home directory, and do not worry if the command looks a little silly. Enter:

chown *daffy daffy*

This command changed the ownership of the directory *daffy* from root to *daffy*. Enter **ls -ld** *daffy* to make sure that it worked.

C3 The next step is to change the group permissions of the home directory. Again from the directory above the home directory

enter the following command (substitute the name of *daffy*'s default group for *newgroup* in the command line):

chgrp *newgroup daffy*

Enter the command

ls -lgd *daffy*

to see if the directory permissions have been changed. The output should look something like this:

```
drwxrwxrwx 1 daffy   newgroup  july 4 12:34 daffy
```

Objective D

Adding the .login File

D1 There is one last thing you must do before *daffy* is actually a member of the system. When a user logs on to the system their *.login* file is read so that the system can set up the different variables necessary to work. (If you are using the Bourne shell, you should create a *.profile* file instead of a *.login* file. If you are using some other shell, or if the new user will be using a graphics workstation, additional files may be required.) There are several ways to set up a *.login* file but we'll just discuss two of them here.

The first strategy is to have a generic *.login* file that is kept permanently in the computer so you can copy it whenever you need to add another user. If you will be adding a number of users this is probably a good idea.

D2 The second strategy is to simply copy your own (not *root*'s) *.login* file to *daffy*'s home directory. Move into *daffy*'s home directory and enter the following line:

cp */usr/you/.login* **.**

Substitute your home directory for */usr/you.*

This command will copy your *.login* file to *daffy*'s home directory. To make sure it worked enter the following command:

ls -a

D3 Just two more quick commands and *daffy* will be an official member of the team. Now that *daffy* has a home directory and a *.login* file, the only thing left to do is to make sure the file will work. Open *.login* using the visual editor and see if there are any variables that contain the path to your home directory. If there are change the path so that it is *daffy*'s home directory.

Now write and quit the file using **:wq**. The only thing left to do is to change permissions and ownership of the file over to *daffy.* Enter the next two commands in the order they appear:

chmod *640 .login*

chown *daffy .login*

Now enter **ls -la** to see if everything looks correct.

Objective E

Assigning a Password

E1 So far we have given *daffy* an entry in the */etc/passwd* file, a home directory and a *.login* file. You are almost done but there is still one piece of unfinished business. At this point *daffy* does not have a password (remember, the password field in the *passwd* file was left empty). As a security precaution you should never have a login without a password, unless you choose to have a guest account set up this way.

To assign a password for *daffy* use the following command:

passwd *daffy*

You will be prompted to enter the new password two times. Remember that a password must be at least six letters long and should contain at least two letters and at least one number.

E2 Now log out of the system and try to log back in as *daffy*. Hopefully you remembered the password you just assigned. Once you have logged in try opening a file and moving around within the system. If everything went right you should have no problems.

E3 Log out again, and log in as the super user.

Objective F

Adding more information

Earlier, we mentioned that the **chfn** command can be used to add information to the miscellaneous field of the */etc/passwd* file. As the super-user, enter the command:

chfn *daffy*

F1 Enter the information that the system requests, pressing RETURN after each entry.

F2 When you return to the shell, you can confirm the information by typing:

finger *daffy*

Any user of the system may now view this information by entering the same command.

F3 Normally, any user may modify his/her own **finger** information with the **chfn** command. If you wish to prevent this, change the permissions of the **chfn** command. Enter:

chmod *go-rx /usr/ucb/chfn*

F4 To restore users' ability to use the **chfn** command, enter:

chmod *go+rx /usr/ucb/chfn*

Objective G
Removing a user

G1 Removing a user from your system is not a difficult procedure. It can be as simple as inserting a word such as *VOID* in the encrypted password field for that user in */etc/passwd*. If the user has created many files that must be saved, you may need to find all files owned by the user and back them up before deleting them.

Before deleting any files, you should first determine who else uses the files and change the ownership of the shared files.

G2 This section introduces the most moderate form of user removal first, and then discusses additional steps that make the removal more extreme.

Objective H
Mild Deletion

H1 The first step in removing a user from your system is to deny the user access to it. The cleanest way to do this is to edit the user's */etc/passwd* entry (using **vipw**) and enter the word *VOID* in the encrypted *password* field. This makes it impossible for any one to login as that user (although that user's files remain unaffected).

H2 Do not leave the *password* field blank. A blank password allows anyone to access the system with that login name and no password.

H3 Do not yet delete the whole */etc/passwd* entry for that user. If you do, you will not only deny the user access to the system but also will affect the files owned by that user. If there is no login name for a file's owner, it is replaced by a number. If you delete a few */etc/passwd* entries you will probably get confused as to what files belong to what ex-user.

Objective I
Backup and selective deletion

I1 Deleting a user's files should be done with care. In general, it is a good idea to back up a user's files before deleting them for two reasons.

- These files may contain information that you may wish to access at a later time.
- These files may currently be used by other users on your system.

I2 To locate all files owned by the user that are not located below the user's home directory the following steps should be observed:

- Find all files belonging to the user, regardless of their location, with the command:

  find / -user *login_name* **-print**

- Back up the files using either **tar** or **cpio**.
- Void the user's password (see the section "Mild deletion" above).

- Before you delete a user's files you should find out whether anyone else is using the files. Use **mail** to ask the other users on the system if they are using the files before deleting them. If any one is using the files, change the ownership and permissions to make sure someone else can access the files.

Once you have all this information at hand, you can proceed deleting the files, copying them to other directories or backing them up.

Objective J
Permanent Deletion

Normally you would have to go through all of the above steps (under **Mild Deletion**) before permanently removing *daffy* from the system. However, since *daffy* is just a practice creation, we know he only owns two files: his home directory, and his *.login* file. (If you have created other files for *daffy*, remove them now.)

J1 Move into *daffy*'s home directory and enter the command:

rm *.login*

J2 Now move up one directory by entering:

cd ..

J3 Remove *daffy*'s home directory with the command:

rmdir *daffy*

J4 Once you have removed all of *daffy*'s files, you may completely remove him from the system by editing the */etc/passwd* file (using **vipw**) and delete *daffy*'s entry.

Objective K

Modifying a User's Environment

K1 There are many different things you can do to change your environment and make it easier to work. One of the easiest ways to modify your environment is to change your *.login* file. (This section assumes you are using the C-Shell.)

K2 Use **vi** to open your *.login* in your home directory:

vi *.login*

Add the following lines to the file substituting your login name where it says *your name*:

```
set path = (/bin /etc /usr/lib /usr/ucb /usr/bin /usr/local)
setenv MAIL /usr/mail/your name
```

Now close the file using **:wq**. Each of these lines tell the computer to do a particular thing. The first line tells the computer to search through the */bin, /etc, /usr/local, /usr/bin,* etc., directories whenever you type in a command. If you keep a directory in your home directory that contains programs you would like to use you can put your home directory in the path so you can access the commands no matter which directory you are located in.

The second line tells **mail** where to put your unread mail. Both of these customizations are helpful but not completely necessary. The next entry will be much more useful in everyday computing.

K3 One of the most useful ways to change your everyday working environment is to add aliases to your *login* file. Aliases allow you to modify the commands you use every day, like **ls** and **cd**. Again open the *.login* file and append the following alias lines complete with spaces.

```
alias lo logout
alias ls "ls -F"
```

The first alias will allow you to simply enter **lo** whenever you want to logout.

The second alias will display a slash (/) after every file that is a directory, and an asterisk (*) after every file that is an executable file.

K4 But before that works you have to do two things. First, close the file *.login* using **:wq**. Then enter the following line:

source *.login*

This will force the shell to read your *.login* file as if you had just logged in.

New User Information Sheet

Login name:_____

Password: (Leave this blank for a new user)

uid (user id number)_____

gid (group id number)_____

Miscellaneous_____

Users home directory (full path name)_____

Start up shell (full path name)_____

Below is the command for finding the highest current **uid** number.

awk -F: '{print $3}' */etc/passwd* **| sort -n | tail -1**

Notes

Notes

System Startup and Shutdown

Introduction

The procedures for starting up and bringing down the UNIX system are automated and require little administrative intervention. Well-informed intervention is necessary, however, when something unusual occurs during the running of one of these automated procedures, or when you're changing standard procedures to accommodate particular needs.

This chapter describes how to use the standard startup and shutdown procedures and introduces concepts for troubleshooting and modifying these procedures.

Prerequisites

Before beginning this module, you should be able to manipulate files from the shell, add and remove user accounts, and create, remove, and move among UNIX directories.

You should also have access to a system that has been assembled and configured, and that has a terminal attached to the console serial port. If you need to find out how to physically set up your system, please look in the System Installation Manual provided with your system.

We are also assuming that UNIX is installed on your fixed disk drive. Usually, systems are shipped with UNIX installed on the disk drive. If not, you will need to look in the System Administrator's Guide (provided with your system) to see how to install UNIX.

Objective

To be able to bring the system up, gracefully shut the system down, perform basic trouble shooting on the startup and shutdown procedures, and implement simple modifications to these procedures.

Procedures

The system console is the terminal plugged into the console port of your UNIX system. The following procedures assume that you are using a regular ASCII (non-graphics) terminal as your system console. If you are using a graphics terminal, you will be able to follow the procedures, and all of the appropriate messages will appear on the screen, but they will appear in "windows" (rectangular boxes) on the screen, and the windows will change in size and appearance several times. Do not let this bother you; the steps for starting up and shutting down the system are the same for ASCII and graphics terminals.

Objective A
Bringing up UNIX

Once your system is installed, plugged in and ready to go, start up your system by performing the following steps:

A1 Turn on the power to the machine.

A2 Press the RESET button, located on the computer front panel, to run the monitor program. If there are two RESET buttons, but you only have one system on the rack, the buttons should be numbered. Press the lower-numbered button.

When you press the RESET button, the bootstrap program is loaded from disk into main memory by a short program located in the startup ROM. The startup ROM is built into the

electronic memory of the computer.

A3 Press [@] on the console terminal when the prompt (:) appears. (On most terminals, [@] is entered by holding down the shift key and pressing the [2] key.)

Pressing [@], or simply waiting about thirty seconds, results in the system performing the following actions:

The UNIX kernel (**/vmunix**) is then executed. The kernel starts several special processes, including the scheduler (**sched**) , and several "daemons" which take care of managing memory and the disk drive.

The **/etc/rc** program, which may call other programs, such as **/etc/rc.local**, or **fsck** to check the consistency of the file systems. If there is a problem with a file system, a message something like this will be displayed:

```
LINK COUNT DIR I=6403  OWNER=root MODE=40770
SIZE=512 MTIME=Nov 24 13:28 1987  COUNT 1 SHOULD BE 2
ADJUST?
```

For more information, see the entry on **fsck** in your UNIX Programmer's Manual.

If a message such as the above should appear, give the system permission to fix the problem. Enter **y** and press [RETURN].

If at the end of the **fsck** run the system informs you that you should reboot the system, press the [RESET] button on the computer's front panel once more, and restart the procedure.

A4 Soon the system comes up and a login banner appears on the console's screen and on any terminals physically connected to your machine, including the console. You are now in *multi-user mode.* Later, in the discussion of the **shutdown** command, we will explain how to move between *single-user mode* and *multi-user mode.*

The prompt

```
login:
```

will appear on your screen.

A5 Enter the appropriate login name, followed by RETURN .

A6 The prompt

```
password:
```

appears on the screen after you've entered the login name. Enter your password followed by RETURN . You are now logged on.

Objective B

Configuring multi-user mode with the /etc/ttys file

B1 When UNIX starts in *multi-user mode*, the **init** process starts an **/etc/getty** (described below) process for each terminal according to that terminal's entry in the **/etc/ttys** file. Graphics workstations are not treated as terminals, so they do not receive an **/etc/getty** process. Each line in the file looks like this:

```
tty##     "/etc/getty label"     ttytype     status
```

The first word, tty##, is the name of the corresponding terminal in the */dev* (device) directory.

The following quoted string is the command to be executed when initializing the terminal before logging in a user. The **/etc/getty** command initializes the terminal and reads a login name from the keyboard. The terminal characteristics (e.g., communications speed) used by **/etc/getty** are determined by the label. For graphics workstations, and other special cases where **/etc/getty** is not used, some other command may appear in this field.

The ttytype is the type of terminal that the system will assume is connected to the corresponding device.

The status will be on if the terminal is to be used as a login terminal. Other keywords may be used in this field also. For more information on **/etc/ttys**, check the entry on **ttys** in section 7 of your UNIX Programmer's Manual.

Note that if you change **/etc/ttys** while the system is running, you need to enter the command:

kill -1 1

to signal the **init** process to reread the file.

Objective C

UNIX system shutdown

When the UNIX system is running, information is constantly flowing between the system's memory and the disk drive. If you were to simply shut off the power to the system while this was going on, you would probably damage some of the data on the disk, possibly crippling your system. The **shutdown** command provides a way to gracefully and harmlessly shut down your system.

C1 Login as **root**.

C2 If you are not in the / directory, type

cd /

C3 Enter the command:

shutdown +1

C4 This command waits a minute before bringing down the system. It also sends a warning message to everyone who is logged in. You can specify the message by including it after the **+1** in the command line; otherwise a default message will be issued.

To specify a different amount of time before shutting down, you can invoke **shutdown** with a different argument:

shutdown +2

will cause the program to wait two minutes before effectively shutting down the system (that is, bringing it down to *single-user mode*).

shutdown now

will shut the system down immediately. This is very bad manners if there are users on the system.

shutdown 16:30

will shut the system down at 4:30 this afternoon. You cannot specify an absolute time any later than 11:59 on the evening of the current calendar day.

C5 The **shutdown** command sends periodic messages to users, warning them that the system is about to be shut down. When the scheduled time comes, **shutdown** brings the system to *single-user* mode.

In single-user mode, only the console terminal is active. From the console, you will have access to a shell with super-user privileges. Also, in *single-user mode*, only a limited portion of the system's files will be available. Some administrative functions, such as repairing file systems and making backups, are best performed when the system is in *single-user mode*.

C6 If you wish to return to *multi-user mode*, press CONTROL-D (that is, hold down the CONTROL key and press D) at the shell prompt in *single-user mode*.

C7 If you wish to power down the system, you may enter the following command in *single-user mode*:

halt

Now wait for the system to inform you that it is Halted, and

shut off the power.

C8 If you simply want to shut the system down, it is possible to bypass *single-user mode.* Enter the **-h** option to the **shutdown** command:

shutdown -h *time*

When shutdown time arrives, the system will halt and you can then shut off the power.

Objective D
Other shutdown options

There are two other options to the **shutdown** command which you may find useful.

D1 The **-r** (reboot) option:

shutdown -r *time*

causes the system to reboot (start *multi-user mode*) from scratch when *time* is reached.

D2 The **-k** (bluffing) option:

shutdown -k *time*

causes the system to continue running uninterrupted when *time* is reached. The usual messages are still sent out, so this option is used to make users *think* the system is going to be shut down.

Conclusion

The health of your UNIX system depends on the orderly start-up and termination of the system's functions. Using the provided utilities and following sound procedures will help prevent major disasters which could inconvenience you or your users.

Notes

Notes

Using the Find Command

Introduction

Directories are very useful for organizing files. Having many directories does, however, have one draw back. Separate directories make it more difficult to find and perform actions on a set of files that are not located within the same directory. The **find** command is designed to reduce this problem by providing a mechanism for finding and performing actions on files located within or below one or more specified directories.

For example, the command

rm *tmp**

will remove all files starting with the letters *tmp* within the current directory. If, however, your goal is to remove all files starting with the letters *tmp* that are within the current directory or any subdirectories of the current directory (and subdirectories of its subdirectories and ...) you need the following **find** command line:

find . -name "*tmp****" -exec rm "{}" \;**

find does not have the prettiest syntax, but nonetheless, it's one of the most useful commands on the UNIX system.

Prerequisites

Before beginning this module you should be able to manage files from the shell, have a basic understanding of the UNIX operating system, be able to set file permissions, and understand the concepts and concerns of being a superuser.

Objective

Upon completion of this module, you will be able to use the **find** command to search for and perform actions on files located within or below one or more specified directories.

Procedures

Many uses of the **find** command will work best if you are logged in as the super user. This is because the **find** command can only search directories that are both readable and executable by you. For this reason we recommend that the procedures in this module be completed while you are logged in as the super user. If this is not an option for you, be prepared to see error messages such as **cannot open** *Directory* or some similar message indicating that you do not have permission to read or execute the named directory.

Objective A
Finding Files and Printing Their Full Pathnames

A1 **Finding a file with a known name:** To find a file with a known name located below your current directory enter the command:

find . -name *filename* **-print**

replacing *filename* with the name of some file located below your current directory (use a file that is not located within the current directory to prove to yourself that **find** is not limited to files within the current directory). This command line instructs **find** to search the current directory and all of its subdirectories (and subdirectories of subdirectories and ...) for the file you specified.

A2 **The general form of the find command:** In general the **find** command is used to search through one or more directories

specified by a *path-list* for a set of files matching one or more *section-criteria* and to perform some *action* on files that match the selection criterion.

These three functions were specified in the above example as follows.

- The *path-list* was **.** specifying that the current directory and all of its subdirectories (and subdirectories of its subdirectories and ...) should be searched.
- The *selection-criterion* was **-name** *filename* which specified that any file with the specified name should receive the action.
- The *action* was **-print** which specified that the full pathname of each occurrence of the file(s) matching the selection criterion should be printed.

The general form of the command line used to specify these functions is:

find *path-list selection-criteria action*

A3 **Finding all files owned by a user:** To find all files on the system owned by a particular user you may need to be logged in as the super user. This is because the **find** command can only search directories that are both readable and executable by you. Become the super user (if you can) and enter the command:

find / -user *username* **-print**

replacing *username* with the login name of some user on your system. The path list for this example is / specifying that the root directory and every directory located below the root directory should be searched. The selection criterion **-user** *username* specifies that all files owned by *username* should receive the action. The action **-print** again specifies that the full pathname of each occurrence of the file(s) matching the selection criterion should be printed.

A4 **Finding all files that have not been accessed within a specified time**: Another useful selection criterion for the **find** command can be used to select all files that have or have not been accessed within a specified number of days. The following command will identify all files that have not been accessed in the last *30* days and have the name *.login*

find / -atime *+30* **-name** *.login* **-print**

Access time is defined as the last time the file was opened for reading. It is updated when the file is **cat**ed, **more**d, **co**pied, etc.

The path list for this example is again / specifying that the root directory and every directory located below the root directory should be searched.

Two selection criteria: **-atime** *+30* and **-name** *.login* specify that all files that have not been accessed within the last *30* days and are named *.login* should receive the action. The plus sign (+) in front of the *30* means *more than 30.* To say "all files that have not been accessed in *30* days, we are actually saying "all files that have been accessed in *more than 30* days." If you had said **-atime** *30* without the plus sign, the message would have been "all files that have been accessed in *exactly 30* days." If you had said **-atime** *-30,* the message would have been "all files that have been accessed in *less than 30* days." We will see more use of the plus and minus sign with numbers in the **find** command later. Note that two selection criteria together imply that both criteria must be met. We will explore the case where one or the other criteria must be met in a later example.

The action in this example is **-print** which says to print the pathnames of files passing the selection criteria.

A5 **Finding all files that have been modified within a specified time**: Like the access time check, modify time can also be used as a criterion for **find**. The following command will identify all files that have been modified in the last *10* days and have the name *.login*

find / **-mtime** *-10* **-name** *.login* **-print**

Modify time is defined as the last time the file was opened for writing. It is updated when the file is written (**:w**) from within **vi**, **touch**ed, etc.

The path list here is /, meaning that all files in root and below should be tested with the selection criteria.

The first selection criterion here is **-mtime** *-10*. It means "all files that have been modified *less than* ten days ago." The minus sign (**-**) is what gives us the *less than*. If we had left out the minus sign before the *10*, we would have been saying "all files that have been modified *exactly* ten days ago," and if we had had *+10* in place of *-10*, we would have been saying "all files that have been modified *more than* ten days ago."

The second criterion, **-name** *.login*, says that the file must be named *.login* for it to receive the action.

The action in this example (because of the **-print**) is to print each pathname passing the criteria.

A6 **Finding all files that have been modified more recently than a specified file**: The above command line can be used to generate a list of files that have been modified within a specified period of time. It is also possible to generate a list of all files that have been modified more recently then a specified file. Try the following command:

find / **-newer** *filename* **-print**

In this **find** command, the path list is /, again specifying that **find** should check all names in the root directory (and in all of root's subdirectories and their subdirectories ...).

The selection criterion in this example is **-newer** *filename*. It means that **find** should only select names for which the most recent modification date (stored with each file) is newer than the modification date for the file, *filename*.

-print specifies the action to be performed in this example for names that pass the selection criterion. It says to print the name.

One practical use of this example is to generate a list a files to be archived onto some backup media. The strategy is to update some dummy file when you create a backup. The modification time on this dummy file is then the same as the last time the system was backed up. The next time you back up the system, this dummy file will have a modification date against which all other file modification dates may be compared. All files that have been modified since the last backup (and thus need to be backed up this time) are listed by the **find** command in our example.

A7 **Identifying the most recent version of a file**: Sometimes you have more than one version of a file, and you want to make sure you are about to print or modify the *most recent* one. Try the following command, substituting the name of one of your files for *filename*:

find / **-name** *filename* **-newer** *filename* **-print**

The path list here is /, meaning root again.

The selection criteria are twofold: First, the **-name** *filename* criterion is used to specify that a match must have the name, *filename*. Only files (or directories) with the name, *filename* will be selected. Second, the **-newer** *filename* criterion specifies that a match must be newer than *filename*. *filename* here resides in the current directory unless a full path is given. Note that the name, *filename*, is the same for both selection criteria; we're looking for newer versions *with the same name* as our current file.

When two selection criteria are used as above, *both* must be satisfied for a name in order for that name to pass the selection test.

The action again (**-print**) is to print the names of the files that pass all of the selection criteria.

If no files are listed by **find**, then we know that the version of *filename* named by the criterion, **-newer** *filename*, is the most recent version of the file with that name. On the other hand, if path names are printed out by **find**, those versions of *filename* have had more recent modifications than the *filename* specified by the **-newer** *filename* criterion.

A8 **Finding all set-user-id files**: Another characteristic of files that **find** can trace is the permission status. Try typing the following command to print the names of files with the set-user-id bit set:

```
find / -perm -4000 -print
```

The path list in the above command is /. Thus, all files in root and that are in root's subdirectories, etc. will be considered in the selection criteria for the command.

The selection criterion for this example is **-perm** *-4000*. This means that only files with their set-user-id bit set should be selected. The **-perm** means we are talking about permissions. The minus sign in front of the *4000* means we are talking about special flags (of which set-user-id is one) rather than ordinary file permissions. And, the *4000* itself, refers to the set-user-id bit in particular.

The action to be performed on each file that matches the selection criteria is just to print the name of the file.

Listing with **ls -l** one of the file names printed by **find** (let's say, */bin/passwd*), should produce something like the following line:

```
-rwsr-xr-x  1 root        20480 Nov  4  1986 /bin/passwd
```

Notice the s in the position of the permissions field where the owner's execute permission is usually specified. This is the

indicator into which **find** tuned when it selected the name, */etc/passwd*.

Note: The set-user-id bit is used primarily for system security. Instead of inheriting its parent process' user id (the default case), the program in a file with the set-user-id bit gets the user id of the file itself.

A9 **Finding all set-group-id files**: The set-group-id bit is similar to the set-user-id bit. Try typing:

find / -perm *-2000* **-print**

Here, by specifying a path list consisting of /, you had **find** check every file in the paths that start with root.

The selection criterion is **-perm** *-2000*, indicating that the file's set-group-id bit should be set. The **-perm** means we are looking at file permissions, the - before the *2000* means we are looking at a special flag in the file permissions, and *2000* refers directly to the set-group-id bit.

The action is to print the name of each such file, and this was done.

One of the file names printed was probably */bin/su*. Listing the permissions of this file should show you that the set-group-id bit is set:

```
-rwsr-xr-x  1 root       20480 Nov  4  1986 /bin/su
```

Note: The security applications of the set-group-id bit are the same as those for the set-user-id bit. Instead of inheriting its parent process' group id (the default case), the program in a file with the set-group-id bit set gets the group id of the file itself.

A10 **Finding all large directories** (over ten blocks in size): **find** also has selection criteria that involve size and that can be open ended. Try the following **find** command to print the names of

all directories of over ten blocks:

find / **-type** *d* **-size** *+10* **-print**

Again, we specified that **find** should check all names in root and that are in directories beneath root. We did this by making root (/) the path list.

Then we gave two selection criteria:

-type *d* First, the type of the entity encountered by **find** must be a *d*irectory. (**find** looks at both file names and directory names.)

-size *+10* Then the directory found must be larger than ten blocks. The *+* in the *+10* means *more than* (more than ten), and blocks are the default size for **find**'s **-size** *num* selection criterion.

The action performed here (due to **-print**) is the printing of the selected path names.

A11 **Finding all large files** (over 138 blocks in size): By modifying the finer options of the previous example, we can restrict find to look at files instead of directories, and we can increase the size to say larger than 138 blocks rather than larger than ten blocks. Try the following **find** command to print the names of all files of over 138 blocks:

find / **-type** *f* **-size** *+138* **-print**

Again, we specified that **find** should check all names in root and that are in directories beneath root. We did this by making root (/) the path list.

Then we gave two selection criteria:

-type *f* First, the type of the entity encountered by **find** must be a *f*ile. (**find** looks at both file names and directory names. Here, we tell it to select the file names.)

-size *+138* Then the file found must be larger than 138 blocks. The *+* in the *+138* means *more than* (more than 138), and blocks are the default size for **find**'s **-size** *num* selection criterion.

The action performed here (because of the **-print** option) is the printing of the selected path names.

Objective B

Using Multiple Selection Criteria

Until now, you have used more than one selection criterion in sequence on a command line to mean that the file should pass *all* criteria for the selection to be made. The following examples illustrate a **find** construct that allows you to specify that the file should pass either one or another selection criterion in order to get the action.

B1 **Finding all set-user-id and set-group-id files**: Two of the above examples might logically be combined into one **find** command. Try the following example:

find / \(-perm *-4000* **-o –perm** *-2000* **\) -print**

In this example, the path list (as usual) is /, so all names in root (and in subdirectories of root and in subdirectories of those ...) will be subjected to the selection criteria.

The selection criteria for this example are **\(-perm** *-4000* **-o** *-2000* **\)**. The parentheses are used here to group **-perm** with its *two* specifications (*-4000* and *-2000*) as one overall criterion. They also keep the selection criteria grouped separately from the **-print** action which follows. The parentheses are preceded by backslashes so they will be shielded from the shell. Without the backslashes, the shell would interpret the parentheses as process grouping characters. The **-perm** means that a file permissions specification is to follow. In this case, there are two permission specifications:

-4000 to test for set-user-id and *-2000* to test for set-group-id. But rather than stringing the *-4000* and *-2000* together sequentially, a **-o** for **or** is inserted between them. Then **find** matches any name having *either* permissions flag set.

The action in this example is still to print the passing file names (**-print**).

So in this command, all names in root or below that have the set-user-id *or* the set-group-id bit set are printed.

B2 **Finding all large files and directories**: The **-o** (for *or*) facility of **find** can be used not only between pieces of a single selection criterion, but also between complete selection criteria themselves. Try the following command:

find / \(-type *d* **-size** *+10* **-o -type** *f* **-size** *+138* **\) -print**

The path list is / so that all names in root and in paths that start with root are considered.

The selection criteria are **-type** *d* and **-size** *+10* together *or* **-type** *f* and **-size** *+138* together. The selection criteria get all names that are directories and are over ten blocks *or* that are files and are over 138 blocks. The *and* (expressed by juxtaposition) takes precedence over the *or* (expressed by **-o**) so there is no need for new parentheses around the two sides of the *or* operator. The parentheses are used to group the selection criteria separate from the **-print**, and the backslashes are again necessary to shield the parentheses from the shell.

The action is again **-print**, so all selected files have their names printed.

Objective C
Performing Actions on Found Files

So far, we have always specified that our selected names should be printed. (We used the **-print** action.) **find** also provides the capability for other actions to take place on found files. Often we want to do a lot more than just print the names.

C1 Changing the group membership of a set of files:

One application of **find** is to change the group membership of a directory and all of its subdirectories etc. Rather than **cd**ing to all of the subdirectories yourself and typing

chgrp *groupname*

in each one, you can execute a **find** command from the highest level directory that you want affected and have it all done in one command.

Try the following command to change the groupname of all the files in and below your current directory.

find . -exec chgrp *groupname* **"{}" \;**

The path list in this command is something new: **.** (dot) which means "the current directory." So **find** considers all files in the current directory and all of its subdirectories (and all of its subdirectories' subdirectories ...).

The selection criterion is empty. That is, for this example, there are no selection criteria specified. This is because we wanted all of the files in the pathlist to be selected for the action. Leaving the selection criteria (which is restrictive) unspecified causes no names to be ruled out, so all are selected.

The action for this example is **-exec chgrp** *groupname***"{}" \;** The **-exec** means that the UNIX command following should be executed on each selected name. In this case, the said UNIX command is

chgrp *groupname* **"{}" \;**

There are a few special characters in this command, because it doesn't look like the typical

chgrp *groupname filename*

command. Rest assured it accomplishes the same thing, though.

"{}" First, the braces ({}) mean that the currently selected filename should be used. That is, whatever filename **find** is considering (after the name passed through the selection criteria) is substituted into the executed command at the position of these braces. The quotation marks around he braces are there to keep the shell from trying to interpret the meaning of the braces. To the shell, the braces have a separate meaning that would interfere with **find**'s meaning. Note: If you are wondering why the writers of **find** had to come up with the {} construct, you may want to read the rest of this note. If you think about it, there is no way to specify what the names of the files will be ahead of time. At the time the **find** command is about to perform its action on a file, a single file name itself has already been selected according to the selection criteria. Therefore, something like the asterisk (*), which gets a substitution of *all names* in the current directory (and all at once), would be ineffective at naming *each* file at the right time. Also, filename substitution for the asterisk takes place at the time of command issuance, when *only the current directory's files* are listed. Therefore, files in subdirectories would be skipped if you were to use the star in the **chgrp** command. Furthermore, it is a fact that in this example you happen to be selecting all files. This is often not the case. When you are selecting out only certain files to have your action performed upon, the asterisk would surely seem inappropriate.

\; Secondly, there is the semicolon. This character is always required at the end of the command part of the **-exec** action. It tells **find** where the command ends. The backslash in front of the semicolon is required to keep it from the shell. If the shell sees it (when there's no backslash), it will think it's a command separator.

So, the result of this **find** command is that for every file and directory *in and under* the current directory, the group is changed to *groupname*.

C2 **Removing *tmp* files:** Users often create temporary files for various and sundry tests. Almost just as often, the user forgets to remove the temporary file right after she or he creates it; maybe she or he needs to keep it around for a while. At some point, there may be a proliferation of temporary files. A user who always creates them with the string *tmp* somewhere in the names of the files has a consistency that will aid in periodically removing all of the files at once.

Try the following **find** command to remove all *tmp* files in or below the current directory. You may want to create some *tmp* files before you try the command, so you can verify that it works.

```
find . -name "*tmp*" -exec rm "{}" \;
```

The path list for this example is **.** (dot) meaning the current directory. So all names in the current directory and in its subdirectories (and in the subdirectories of its subdirectories ...) are subjected to the selection criteria.

The selection criterion is one we've seen before: **-name "*tmp*"**. The ***tmp*** refers to any filename that has the string, *tmp*, in it. The quotation marks around it are required to shield the asterisks from the shell. If the shell saw them, it would try to match all the filenames at once, and furthermore, they'd only be from the current directory. This **find** command is thus looking for any name in or below the current directory

that has the string, *tmp*, in it.

The action is another **-exec**. It does an **rm** of *filename*, where *filename* is the currently selected name from the selection criterion. The {} are used in the command line to represent this current name. Remember that the quotation marks around it are required to shield it from the shell, and the \; at the end of the line is used to end the command part of the **-exec**.

C3 **Changing the owner of all files belonging to a particular user**: Should you find it necessary to rearrange the ownership of certain files on your system, the following command will be an aid.

find *path* **-user** *olduid* **-exec chown** *username* **"{}"** **\;**

In this **find** command, *path* is the path list, so all of the files in *path* or in paths beginning with *path* are subjected to the selection criterion.

The selection criterion here is **-user** *olduid*. It means that the name should be checked to see if its user id (or owner) is *olduid*, where *olduid* is a user id number. All names that have the user id, *olduid*, are selected for the action.

The action in this case is **-exec chown** *username* **"{}"** **\;**. It executes the **chown** command to change the ownership of the selected file to *username*. Again, the quotation marks are necessary to shield the braces from the shell, and the semicolon ends the command part of the **-exec**. The semicolon is escaped with a backslash so that it, too, will not be interpreted by the shell.

This command changes the ownership of every file and directory previously owned by *olduid* in or below root to *username*.

C4 **Searching for a pattern in all files below a specified directory**: When writing programs or manuscripts, the need often arises for a way to search all files in and below a certain directory for a certain string. A **grep** command alone will only

handle the files in one directory, but a **find** command that does a **-exec grep** might get the job done. Try the following command:

find *path-list* **-print -exec grep** *pattern* **"{}" \;**

Here, there is no selection criteria to limit what files are selected, so all files in and below *path-list* are selected. The actions are twofold this time. First, the name of the file is printed due to the **-print** action. Secondly, the **-exec** action does a **grep** for *pattern* in the file. The reason for the **-print** is so that you will be able to see what files the **grep** results are referring to. When given a single file name, the name of the file is not printed by **grep**, so we have **find**'s **-print** do it for us.

Conclusion

After trying many examples, you can now see the usefulness of **find**. The general form of the command line used to specify **find**'s functions is:

find *path-list selection-criteria action*

Its path list can be any list of one or more directories, its selection criteria look at nearly any aspect of a file or a directory, and its action section can be used to print the selected name or to execute just about any UNIX command on the file.

Notes

Notes