April 1996 • VOL. 2 NO. 4
US $11.50

INSIDE SOLARIS™
Tips & techniques for users of SunSoft Solaris™

in this issue

Automating common ftp tasks

By Marco C. Mason

Suppose that you manage a large network where multiple machines need to have a local copy of some set of files, and you're frequently updating these same files. You start `ftp` and transfer the files like this (what you type is in color):

```
$ ftp widget1.cobb.zd.com
Connected to widget1.
220 widget1 FTP server (UNIX(r) System
   V Release 4.0) ready.
Name (Widget1:marco): anonymous
331 Guest login ok, send ident as password.
Password:
ftp> cd /pub/data
250 CWD command successful.
ftp> put daily.summary
200 PORT command successful.
150 ASCII data connection for
   daily.summary (140.244.96.202,32817)
226 Transfer complete.
local: daily.summary remote: daily.summary
21468 bytes sent in 11.32 seconds (1.9
Kbytes/s)
ftp>
```

Ugh! You've sent only one file, and you've barely gotten started. If you need to send a few files in different subdirectories to a dozen machines, you'll be baby-sitting `ftp` for quite a while. Wouldn't it be better if a command could do all the transfers for you?

In this article, we'll show you how to create macros to make large file transfer jobs simpler using `ftp`. We'll also show you how to make `ftp` automatically log you in when you specify the computer to which you want to connect.

Creating macros

The first thing we can do to make `ftp` simpler to use is to create macros for common tasks. You can create your own commands in `ftp` by building them out of other `ftp` commands. These new commands are called macros. You create your commands with the `macdef` command. To use the `macdef` command, all you need to do is type *macdef* and follow it with the name of your new macro. Then, type each command you want your new macro to perform. When you're finished, enter a blank line to tell `ftp` that your macro is complete.

Figure A shows an example macro, named transfer, that will copy our daily summary files to a remote machine. Please note that this macro assumes that we're already connected to the machine.

Figure A

```
macdef transfer
cd /pub/data
put daily.summary
put graphs/daily.orders
put graphs /daily.inquiries
put graphs /daily.cancels
cd /pub/spreadsheets
put daily.worksheet.wks
<blank line>
```

We use this macro to transfer our daily summary files to a remote machine.

Executing macros

Now that we have our macro, transferring a set of files from our machine to a remote machine is simpler. Once we're connected, all we have to do is tell `ftp` to execute the new macro. You do so by entering *$macroname* at an `ftp` prompt, like this:

```
ftp> $transfer
```

INSIDE SOLARIS™
Tips & techniques for users of SunSoft Solaris

Inside Solaris (ISSN 1081-3314) is published monthly by The Cobb Group.

Prices	Domestic	$115/yr ($11.50 each)
	Outside US	$135/yr ($16.95 each)
Phone	US toll free	(800) 223-8720
	UK toll free	(0800) 961897
	Local	(502) 493-3300
	Customer Relations fax	(502) 491-8050
	Editorial Department fax	(502) 491-3433
	Editor-in-Chief	(502) 493-3204

Address Send your tips, special requests, and other correspondence to:

The Editor, *Inside Solaris*
9420 Bunsen Parkway, Suite 300
Louisville, KY 40220
Internet: inside_solaris@merlin.cobb.zd.com.

For subscriptions, fulfillment questions, and requests for bulk orders, address your letters to:

Customer Relations
9420 Bunsen Parkway, Suite 300
Louisville, KY 40220
Internet: cr@merlin.cobb.zd.com.

Postmaster Second class postage is paid in Louisville, KY.

Send address changes to:

Inside Solaris
P.O. Box 35160
Louisville, KY 40232

Advertising For information about advertising in Cobb Group journals, contact Tracee Bell Troutt at (800) 223-8720, extension 430.

Back Issues To order back issues, call Customer Relations at (800) 223-8720. Back issues cost $11.50 each, $16.95 outside the US. We accept MasterCard, VISA, Discover, or American Express, or we can bill you.

Staff	Editor-in-Chief	Marco C. Mason
	Contributing Editor	Jeff Warner
		Al Alexander
	Editors	Linda Recktenwald
		Martha Bundy
	Production Artist	Margueriete E.Winburn
	Circulation Manager	Mike Schroeder
	Editorial Director	Linda Baughman
	Publisher	Lou Armstrong
	President/CEO	J. Thomas Cottingham

Advanced macro operations

Even though transferring all the files to a single remote computer is simple, we still must connect to each machine that we need to update and then execute our macro command. Fortunately, there's a better way. You can write a macro that accepts arguments. In other words, you can write a macro so that when you execute it like this

```
ftp> $send_files widget1 widget2 arthur
```

your macro has the ability to read the arguments widget1, widget2, and arthur. To do so, when you write your macro, use the character sequence `$1` to refer to the first argument, `$2` to refer to the second, and so on. Therefore, we could write the send_files macro to send our files to the machines widget1, widget2, and arthur, as shown in Figure B.

Please note that each argument is separated by spaces, unless you use quotes. If you use quotes, you may embed spaces in your arguments. Thus `$transfer "arg 1" "arg 2"` has only two arguments.

Figure B

```
macdef send_files
open $1
$transfer
close
open $2
$transfer
close
open $3
$transfer
close
<blank line>
```

This version of the send_files macro will send our daily summaries to three machines.

This method is a lot better. Now you can start `ftp`, issue the send_files macro with the list of machines you want to send the files to, and walk away. No more baby-sitting the computer.

However, this macro could still be improved. First, it's repetitious—whenever you do repetitive tasks on a computer, there's usually a better way. (Second, what happens if you want to transfer the daily summaries to more or fewer than three machines?

Both of these problems can be solved by another special character sequence: `$i`. When you use `$i` in a macro definition, you're telling `ftp` to execute the macro once for each argument. The first time `ftp` executes the macro, `$i` refers to the first argument. The second time it executes the macro, `$i` refers to the second argument, and so on, until there are no arguments left. Thus, we can rewrite our send_files macro as shown in Figure C. Using our improved send_files macro, we can transfer our daily summaries to as many machines as we want, simply and easily.

Figure C

```
macdef send_files
open $i
$transfer
close
<blank line>
```

The improved version of the send_files macro will send our daily summaries to any number of machines.

Special notes

Of course, not all is wine and roses. When you're using macros with `ftp`, you're limited to having only 16 defined at one time. Another limitation is that all your macros combined must have less than 4,096 characters.

In addition, since the `$` symbol is used in macro definitions, you can't blithely use it whenever you want one in your macros. In order to put a $ in your macro definitions, you must precede the $ with a \. To put a \ in your macro definitions, you must use \\.

Storing macros for future use

While macros are a very nice feature, why use them if you have to redefine them each time you start the `ftp` program? Fortunately, the designers had similar thoughts: When you start `ftp`, it looks in your *$HOME* directory for a file named *.netrc*. If it finds this file, it will automatically scan this file for macro definitions and load them into memory. By the time you get to the first prompt, all the macro definitions in the *.netrc* file are ready to use.

You can put a macro in your *.netrc* file by entering it just as you would at the `ftp` prompt. Don't forget to place a blank line after the last line in the macro to end the macro!

The init macro

Do you find yourself executing the same set of commands each time you run `ftp`? If so, you'll like one of the other benefits of using the *.netrc* file. It turns out that when `ftp` loads all the macros in the *.netrc* file, if it finds one named init, `ftp` executes it before giving you control.

For example, when you transmit a large file, do you enjoy staring at the screen waiting for the transfer to complete? Neither do we. When we start a file transfer, we open a new terminal window and work on a different project.

Just so we can tell how things are progressing, we always execute the `hash` command so that `ftp` will print hash marks (#) as it transfers the file. This way, we can easily look up from our job to see if `ftp` is still banging away. Similarly, we execute the `bell` command so that `ftp` beeps us when the transfer is complete. This allows us to take a break from our other project and start the next file transfer operation.

Rather than execute the `bell` and `hash` commands each time we start `ftp`, we created the init macro, shown in Figure D, in our *.netrc* file to execute them for us.

Figure D

```
macdef init
bell
hash
<blank line>
```

By putting the `bell` and `hash` commands in our init macro, we'll never have to execute them manually again.

Automatic login

When you examine Figures B and C, you might think we made an error. After all, the `open` command doesn't automatically log you in, does it? By itself, it doesn't. But if you put the machine definition information into your *.netrc* file, it can. This is a great feature even if you're not using macros, as it can help you log in to machines without having to type in your user name and password each time.

When you start `ftp` with a host name on the command line like this

```
ftp widget1.cobb.zd.com
```

or if you simply try to open a connection from an `ftp` prompt, like this

```
ftp> open widget1.cobb.zd.com
```

`ftp` scans its list of machine definitions and, if it finds the specified machine in its list, it will provide whatever information it has. If you specify all the information required, then `ftp` will automatically log you in without prompting you for the user name and password.

You can create a machine definition by putting the following text in your *.netrc* file:

```
machine machine_name
login user_name
password password
```

All you need to do is replace the italicized text with the required information. You can put the machine, login, and password clauses on a single line if you prefer.

Combining all the concepts we've talked about, we created a *.netrc* file, shown in Figure E on page 4, that contains our macros and machine names so we can perform all our daily summary updates just by starting `ftp` and typing

```
$transfer widget1 widget2 arthur
```

Temporarily disabling ftp's automatic login feature

Occasionally, you may need to log in to a computer with a different account than the one described in your *.netrc* file. In this case, you need a way to tell `ftp` to ignore the machine definition information in your *.netrc* file but to otherwise run normally.

You can do this by starting `ftp` with the `-n` option. This tells `ftp` to read only the macros from the *.netrc* file and to ignore the machine definitions. When you use the `-n` option, you'll have to log in to each machine manually. To start `ftp` using this switch, just type

```
ftp -n hostname
```

Protect yourself!

All the features provided by the *.netrc* command are great time-savers. However, there's one security problem that you need to be aware of. If you use the automatic login feature with a real account (i.e., not an anonymous login), then anyone who can read your *.netrc* file can find out your user name and password on that machine.

You can plug this security hole by telling Solaris not to let anyone read the file but you. You can do so by using the command

```
chmod go-r .netrc
```

This command tells Solaris to remove the read permission from other accounts inside and outside of your group. Now no one may read your *.netrc* file but you, the root, and the superuser accounts.

Figure E

```
machine widget1 login anonymous password marco@widget1
machine widget2 login anonymous password marco@widget1

machine arthur
login marco
password splemliferous

macdef init
bell
hash

macdef transfer
cd /pub/data
put daily.summary
put graphs/daily.orders
put graphs /daily.inquiries
put graphs /daily.cancels
cd /pub/spreadsheets
put daily.worksheet.wks

macdef send_files
open $i
$transfer
close
```

Our .netrc *file allows us to automatically log in to three machines and to transfer our daily summaries.*

Conclusion

As you can see, `ftp` has some very powerful features that let you customize its operation. If you take full advantage of the macro facilities and the machine-definition features, you can simplify your daily chores considerably. ❖

Marco C. Mason is a freelance computer consultant and author based in Louisville, Kentucky. He's worked on cattle feeding systems, automated destructive equipment testing, and the largest computer-controlled sound system in the world.

software installation

Installing the NCSA WWW Server

By Al Alexander

For the last several years, the Internet and the World Wide Web have been the most publicized and possibly the most exciting part of computing. It seems that every company, large and small alike, is putting its corporate information on the Web.

The technology has recently spread even further to create corporate intranets—in-house Web servers that store information that can be accessed by employees on internal LANs. Intranet information includes employee manuals, telephone directories, policy manuals, engineering documents, and much more. The Internet and intranets promise to become even more exciting as technologies such as VRML and Java bring animation to the Web.

Since the inception of the Web, NCSA (National Center for Supercomputing Applications at the University of Illinois) and CERN (Center for European Particle Physics Laboratory) have created free, public domain Web server software known as HTTP (HyperText Transfer Protocol) daemons.

These HTTP daemons allow schools, corporations, and government agencies to create their own Web servers on UNIX platforms. As the Web has evolved over the last two years, NCSA's Web server software has won the battle for the public domain HTTP daemon Web server, offering more features and functionality than the CERN version.

In this article, we'll explore the process of acquiring NCSA's HTTPD Server software and installing it on a Solaris 2.4 system. In the process, we'll set up a fictitious Internet Web site that we'll call *www.alexander.com*.

Obtaining the NCSA HTTPD Server software

The first part of the installation process is obtaining the NCSA HTTPD Server software. The Server software can be obtained directly from NCSA via the Internet.

The easiest way to retrieve the file is to point your Web browser (i.e., Netscape Navigator, Mosaic, or a similar product) to the URL *http://hoohoo.ncsa.uiuc.edu/docs/setup/PreCompiled.html*. This URL displays a self-explanatory Web page that allows you to download the entire NCSA Server software distribution. This distribution includes binary executables and necessary subdirectory structures but, unfortunately, no installation instructions.

At the time of this writing, the most recent version of the NSCA HTTPD Server software is 1.5a. It's available for the following Sun platforms:

- SunOS 5.4 / Solaris 2.4 SPARC
- SunOS 5.3 / Solaris 2.3 SPARC
- SunOS 5.4 / Solaris 2.4 x86 (Intel)
- SunOS 4.1.3 / Solaris 1.x SPARC

Putting the Server software in an appropriate directory

The default directory location for the Server distribution is */usr/local/etc/httpd*. You can change this default as desired, but it fits our directory tree structure very well, so we'll install it at that location.

The file that you'll download from NCSA's Web site has been `tar`ed and `compress`ed. For the Solaris 2.4 x86 environment, the file we downloaded, named *httpd_1.5a-export_solaris2.4_x86.tar.Z*, was 498,103 bytes, or about 0.5MB.

After downloading the file, place it in a directory named */usr/local/etc*. If you don't have this directory already, create it as follows:

```
mkdir -p /usr/local/etc
```

Next, copy the file to that directory, and then make that your current working directory:

```
cp <filename> /usr/local/etc
cd /usr/local/etc
```

As we mentioned, the distribution is `tar`ed and `compress`ed. Therefore, the next step is to uncompress the file:

```
uncompress <filename>
```

This expands the x86 distribution to a 1,130,496-byte file. Next, use the `tar` command to extract the `tar` archive:

```
tar xvf <filename>
```

This command extracts all of the NCSA HTTPD Server files into a subdirectory named *httpd_1.5a-export*. The last step in the installation is to rename that directory to the default name *httpd*:

```
mv httpd_1.5a-export httpd
```

The NCSA distribution is now installed in the default location, */usr/local/etc/httpd*. Move into the *httpd* directory and list the files in that directory:

```
cd httpd
ls -al
```

The complete distribution includes not only all of the necessary binaries and configuration subdirectories and files, but also the source code for the HTTPD Server. If you're a C programmer with a C language compiler, you'll find this code valuable if you want to make changes to the actual binary.

The directory */usr/local/etc/httpd* is called the ServerRoot directory. At a minimum, the files and directories shown in Table A should be in the ServerRoot directory.

Table A

File or Directory	Description
httpd	NCSA HTTPD Server binary executable file
./conf	Configuration file subdirectory
./logs	Server log files subdirectory
./support	Various support-related programs subdirectory
./cgi-bin	Common Gateway Interface files subdirectory

These files and directories should be in the ServerRoot directory.

The *logs* directory didn't exist in the distribution, so you must create it manually:

```
mkdir logs
```

Other files may also be in your ServerRoot directory, including *BUGS*, *CHANGES*, *COPYRIGHTS*, *CREDITS*, a Makefile, and a README. You may want to view these files to understand more about the distribution contents.

Modifying the Server configuration files

You can configure the NCSA HTTPD Server in three different ways:

1. At compile time by modifying the C language source code
2. Through startup configuration files
3. Through run-time configuration

The first option requires a C language compiler and is rather inflexible when you want to make changes to the Server as it's running. The third option is typically used to control access to portions of directory trees.

The second option, using configuration files, is probably the most common method of configuring the NCSA Server. Throughout the remainder of our discussion, we'll examine the second method and show you how to configure the NCSA Server by modifying its startup configuration files.

The *conf* subdirectory contains the Server configuration files. When the Server daemon starts, it examines three files in this directory, as shown in Table B

Table B

Filename	Description
access.conf	Access control file
httpd.conf	Server configuration file
srm.conf	Resource configuration file

These configuration files tell the HTTPD Server the information specific to your location.

When you look in your *conf* subdirectory, you'll notice that these three files exist in the *r* subdirectory, but they're followed by the extension *-dist*. These files are sample files that come with the distribution.

Copy these three *-dist* files to the names shown above, and then edit them to customize them for our environment:

```
cd conf
cp access.conf-dist access.conf
cp httpd.conf-dist httpd.conf
cp srm.conf-dist srm.conf
```

By copying these files instead of renaming them, we keep the original files as a future reference.

Each of the three files contains a fair number of comments. Commented lines are those lines that begin with the # symbol. We'll look first at the *access.conf* global access configuration file.

access.conf

The file *access.conf* is the global configuration file. This file contains only a few variables, and if you put your Server files in the default locations, the defaults for these variables will be fine.

Please note that if you decide to put your Server or your Home Page documents in alternate directories, you'll need to change the two Directory entries in this file to reflect your specific locations. Because we're installing the Server files in the default locations, no changes are required to the *access.conf* file.

httpd.conf

The main Server configuration file is *httpd.conf*. When it starts, the Server daemon looks in this file to determine important information, such as the TCP/IP port to run on, which user and group will own the Server daemon, etc.

For most basic HTTPD Server installations, many of the default values will work just fine. The few options that may need to be changed initially are shown in Table C.

Table C

Variable Name	Default Value
ServerName	<none>
ServerAdmin	you@your.address
ServerRoot	/usr/local/etc/httpd
User	nobody
Group	-1 VirtualHost

These are the configuration options in the httpd.conf *file that you'll probably want to customize.*

You should set the ServerName variable to the TCP/IP name of your UNIX server in your */etc/hosts* file (or NIS database). Since we're setting up a site named *www.alexander.com*, we'll use the following statement:

```
ServerName www.alexander.com
```

You should change the variable ServerAdmin to the E-mail address of the person administering your Web site. For the purposes of this article, we'll assume that the Web administrator's username is al, so you should set the ServerAdmin variable as follows:

```
ServerAdmin al@www.alexander.com
```

As we mentioned, ServerRoot is the root, or top, level directory of the HTTPD Server files and directories. Because our Server is installed in the default location, there's no need to change this variable. Please note that if the Server was installed into another directory, such as */var/ncsa_httpd*, you'd need to modify the ServerRoot variable as follows:

```
ServerRoot /var/ncsa_httpd
```

The User and Group variables represent the owner of the Server daemon while it's running on the computer system. Although the root user must start the HTTPD Server, the ownership of the child processes changes to the User and Group values specified in the *httpd.conf* file. This is done to enhance security by allowing you to make the daemon run with fewer privileges.

The User and Group default values are nobody and -1, respectively. These are good choices, but a minor change may be required. On a typical Solaris machine, a username of nobody already exists, so no change is required there.

A group named nobody also exists, but the default group ID for the group nobody on Solaris 2.4 is 60001, not -1. Therefore, you need to change the Group variable to the name nobody instead of using the number -1, as shown below:

```
User nobody
Group nobody
```

The VirtualHost variable (the last few lines of the file) is configured to allow your Server to respond to client requests for multiple hostnames. For instance, it's possible to configure your Server to respond not only to requests for *www.alexander.com*, but also for other domain names, such as *www.acme.com* and *www.foo.com*.

This option is typically used by Internet service providers who sell Internet access to corporations, but not by corporations, schools, or government agencies. The best thing to do with the VirtualHost variable is to comment out the lines that configure it:

```
# <VirtualHost 127.0.0.1 Optional>
# DocumentRoot /local
# ServerName localhost.ncsa.uiuc.edu
# ResourceConfig conf/localhost_srm.conf
# </VirtualHost>
```

One final comment on the *httpd.conf* file: Please note that the HTTPD Server listens on TCP/IP Port 80 by default. This is the default port that Internet browsers such as Netscape call on when looking for a Web server. The HTTPD Server must be configured to listen on the same port that the browser is calling, so we'll leave the Server at port 80. (If you change the port number, no one will be able to access your Web server without changing their Web browser software.)

srm.conf

The file *srm.conf* configures the server's resources. This configuration file defines the name space that users of your Server can see. You may need to change several of the default variables in this file to configure your site. These variables are shown below:

```
DocumentRoot /usr/local/etc/httpd/htdocs
Redirect /HTTPd/http://hoohoo.ncsa.uiuc.edu/
```

The DocumentRoot variable defines the location of your home page documents. By default, your initial home page document—*index.html*—and the remainder of your HTML documents must be in the */usr/local/etc/httpd/htdocs* directory and subdirectories of that directory. If you want to keep your home page documents in another location, you'll need to change the DocumentRoot, Alias, and ScriptAlias variables to reflect that location.

You don't need the Redirect variable for an initial installation, so you can comment it out:

```
# Redirect /HTTPd/http://hoohoo.ncsa.uiuc.edu/
```

Because we're configuring our server to work in the default location, we'll use the default directory. First, create the *htdocs* directory in the */usr/local/etc/httpd* directory:

```
mkdir /usr/local/etc/httpd/htdocs
```

Let's create a home page!

Now we nearly have the Web server configured. Before we proceed to the last step, that of running it, let's create a home page. To do so, go to the *DocumentRoot* directory by typing

```
cd /usr/local/etc/httpd/htdocs
```

Now, using `vi` (or a similar editor) create a text file named *index.html*. Make sure it contains the following text. It's very important that you copy this text *exactly.*

```
<HTML>
<TITLE>My Home Page</title>
<h1>Welcome to my Home Page!</h1>
<h2>Documents on the system include:</h2>
<ul>
<li><a href="http://www.alexander.com/
bio.html">
A description of the site manager.</a>
<li>Watch for new documents in the future!
</html>
```

Figure A

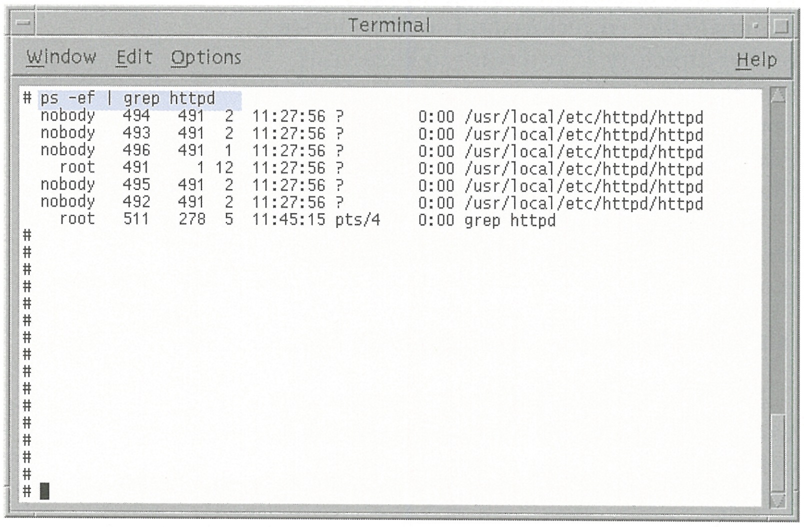
```
httpd Usage:

httpd [-d directory] [-f file] [-v]

-d directory    specify an alternate initial
                ServerRoot directory
-f file         specify an alternate Server
                Configuration file
                (httpd.conf)
-v              display version information
```

You may use these options to start the HTTPD daemon with alternate parameters.

Figure B



```
# ps -ef | grep httpd
  nobody   494   491  2  11:27:56 ?        0:00 /usr/local/etc/httpd/httpd
  nobody   493   491  2  11:27:56 ?        0:00 /usr/local/etc/httpd/httpd
  nobody   496   491  1  11:27:56 ?        0:00 /usr/local/etc/httpd/httpd
    root   491     1 12  11:27:56 ?        0:00 /usr/local/etc/httpd/httpd
  nobody   495   491  2  11:27:56 ?        0:00 /usr/local/etc/httpd/httpd
  nobody   492   491  2  11:27:56 ?        0:00 /usr/local/etc/httpd/httpd
    root   511   278  5  11:45:15 pts/4    0:00 grep httpd
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
# █
```

You can view the processes running the Web server by typing ps -ef | grep httpd *at the command prompt.*

Now, in our fictitious example, any time a user points a browser at the URL *http://www.alexander.com,* he or she will receive this document as the home page. After you've created your home page, return to the *conf* directory to finish changing the *srm.conf* file:

```
cd ../conf
```

Starting the HTTPD Server daemon

Now that you've properly modified the configuration files, you can start the HTTPD Server daemon in one of three different ways:

- in the */etc/inetd.conf* file
- from the command line
- from a startup shell script

For performance reasons, you shouldn't start the server from the */etc/inetd.conf* file. There's a lot of overhead involved in starting the HTTPD daemon in this manner that slows the response time under a heavy load.

Therefore, we'll initially start our server from the command line. Once it's working, we'll create a simple shell program that will start the server every time the computer boots into multi-user mode.

The *httpd* file in the *ServerRoot* directory (*/usr/local/etc/httpd/httpd*) is the server binary executable file. When this file runs, it reads the configuration files in the *conf* subdirectory and begins running as a daemon process. Usage options for starting the HTTPD daemon from the command line are shown in Figure A.

Because the server is installed in the default location, no options are required at the command line. Therefore, to start the server, type

```
/usr/local/etc/httpd/httpd
```

You shouldn't see any error messages when the server starts. If you do see messages, record those messages and carefully review the previous steps.

Also, note that the server automatically runs in the background. You don't need to use the & character after the command to run it in the background.

If everything is working properly, you should be able to identify six processes running on your computer waiting for client requests, as shown in Figure B. Running the highlighted command in Figure B should return seven output lines. (The last output line in Figure B shows the process that's running the **grep** command that's filtering out the processes dealing with *httpd*.)

The output displays the process you created, process ID (PID) 491, owned by root, and five other processes, owned by the user nobody. The five processes owned by nobody are child processes of the parent process, PID 491.

You created five child processes because the variable StartServers is set to 5 in the *httpd.conf* file. These five processes listen on port 80 for client requests and respond to those requests as they occur.

The NCSA HTTPD Server is now running on your Solaris server. To verify that an end user can actually see your home page, you'll need access to another computer on your TCP/IP network that has a browser, and you must point that browser at your new Web server.

In our fictitious example, the server is named *www.alexander.com*. On another computer on my network that has a browser, we'd enter the URL *http://www.alexander.com*. The home page document we created above would appear. Figure C shows how Netscape would display the home page for *www.alexander.com*.

Starting the server automatically at boot time

With the server starting properly from the command line, the last thing you need to do is to start the server processes automatically at boot time. You can easily accomplish this by creating a shell program and putting that shell program in the */etc/rc2.d* directory.

Figure D shows a simple shell program to accomplish this task. The shell program is named *S99ncsa_httpd*, which is consistent with the naming structure in the */etc/rc2.d* directory.

VERY IMPORTANT NOTE: Make sure you test this shell program from the command line before putting it in the */etc/rc2.d* directory and rebooting the computer. If there's an error in this file and the computer can't get past it during the boot process, your computer could hang at boot time!

First, create this file in the */tmp* directory. Then kill the old server processes by killing the parent process (PID 491 in our example):

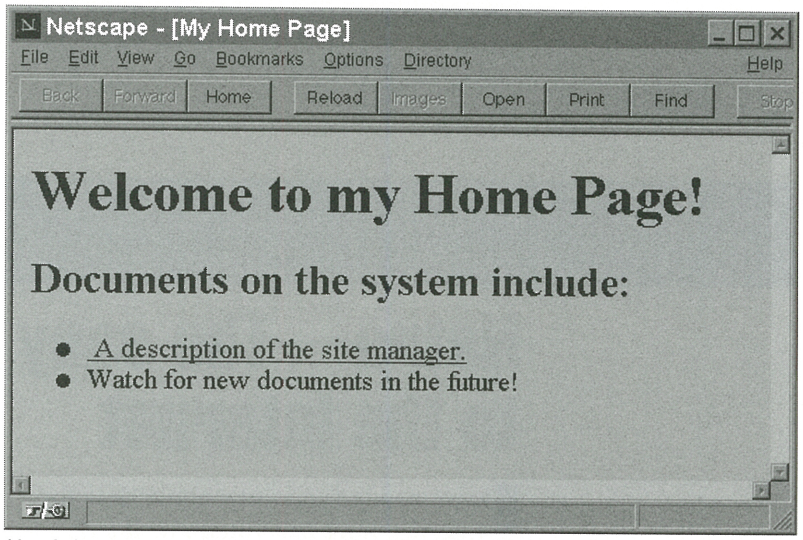
```
kill 491
```

(To find the parent PID #, just look at the output of the `ps -ef | grep httpd` commands, and look for the line starting with root. The number following root is the PID of the parent's process. Killing this process will automatically kill the child processes as well.)

Now that you've stopped the Web server, you can test the startup shell script by typing the following command:

```
sh ./S99ncsa_httpd
```

Figure C


Netscape - [My Home Page]
File Edit View Go Bookmarks Options Directory Help
Back Forward Home Reload Images Open Print Find Stop

Welcome to my Home Page!

Documents on the system include:

- A description of the site manager.
- Watch for new documents in the future!

Here's how Netscape running under Windows 95 interprets our new home page.

Figure D

```
#!/bin/sh
#
# PROGRAM: S99ncsa_httpd
#
# PURPOSE: Start the NCSA HTTPD Server daemon at boot time.
#
#
# Run the NCSA HTTPD Server command:
#

  /usr/local/etc/httpd/httpd

#
# Check the Exit Status of the previous command.
#   If the status = 0, it started ok.
#   If the status != 0, it did not start correctly.
#

  STATUS=$?

  if [ $STATUS -eq 0 ]
  then
   echo "NCSA HTTPD Server Daemon started successfully."
  else
   echo "The NCSA HTTPD Server Daemon did not start properly."
  fi
```

This script starts the NCSA Web server daemon and then tells you whether it started correctly.

If the server is started properly, you'll see the message *NCSA HTTPD Server Daemon started successfully*. If it fails, or if it hangs for any reason, you'll need to begin the debugging process.

You can verify that the server is running by reissuing the `ps -ef | grep httpd` command and verify that it shows a parent process and five child processes. If the startup shell script works successfully, copy it to the */etc/rc2.d* directory:

```
cp /tmp/S99ncsa_httpd /etc/rc2.d
```

The Solaris operating system will now automatically run this script every time your computer is booted to run state 2 (the normal multi-user mode).

Conclusion

The Internet and intranets have helped to bring an enormous amount of information to end users in the last few years. Configuring the NCSA Web Server is a relatively simple and straightforward process that can help your organization achieve further information distribution goals that can benefit everyone at your company. ❖

Alvin J. Alexander is an independent consultant specializing in UNIX and the Internet. He has worked on UNIX networks to support the Space Shuttle, international clients, and various Internet Service providers. He provided UNIX and Internet training to over 400 clients in the last three years.

connecting to the Internet

Netra—The easy way to connect to the Internet

If you need to connect a network to the Internet, how will you do it? If FTP, Archie, WWW, and Mosaic are foreign words to you, you'll probably want to investigate Sun Microsystems' Netra series of Internet servers. Sun's Netra series is a line of computers preconfigured to act as Internet servers. These servers are designed to connect your network to the Internet with a minimum of fuss.

Even if connecting to the Internet is nothing new to you, you still may want to become familiar with the Netra series. After all, other departments in your company may want access, and you may not want to be stuck with managing them. Or perhaps a less-savvy friend in another company would like a little help.

What is a Netra Internet server?

The Netra line of Internet servers is a series of Sun computers designed from the ground up to connect to the Internet. All the software you need comes preinstalled on the computer, so you don't have a complex configuration job facing you. The hard parts are done by your local Sun distributor.

The Netra Internet servers come in several sizes. The smallest of these is designed to serve from one to 20 users, and the largest handles over 200 users.

What does it buy you?

If you get a Netra box for your network, the following software is already configured for use:

- Solaris version 2.4 with revised PPP drivers
- Automatic performance tuning daemon
- DNS client and server (to help manage IP addresses)
- Automated PPP, Token Ring, Ethernet, and ISDN configuration (for access to the Internet or remote clients)
- POP2, POP3, and IMAP protocol (for handling E-mail)
- HTML-based tools for administration (allows you to use a Web browser for administration)

- Telnet and FTP (for accessing remote systems)
- NCSA WWW server and Netscape Communications Server (provides access to multimedia WWW pages)
- Anonymous FTP server (allows you to make documents available for people on the other side of the firewall)
- Optional Solstice FireWall-1 software (protects your entire LAN from hackers on the Internet)

Summary

If you've been given the task of connecting your server to the Internet, there's a quick and easy way to do it. If you don't have the time to learn all the configuration management tasks yourself, you can use the Netra series to get you online right away. ❖

An introduction to shell scripts

By Marco Mason

If you're like most people, you often find yourself repeating tasks. Each time, you must type the same series of commands to get a job done. Wouldn't it be nice if you could just keep a list of commands in a file and tell the computer to execute the whole list?

It turns out that you can do this and much more with shell scripts. Shell scripting allows you to perform complex tasks like looping, decision-making, and math. However, you don't have to learn everything about shell scripting all at once. In this article, we're going to introduce you to the bare essentials: We'll show you how to turn a list of commands that you want to execute into a shell script.

A very simple script

Just for fun, let's create a simple shell script that will find all the files ending with the extension *.bak* in your home directory and all subdirectories below it. To make it more interesting, we'll find only those files that are over a week old and haven't been accessed in that time. After reading the `man` page for the `find` command, you'd figure out that you should type something like this:

```
find $HOME -name '*.bak' -atime +7 -mtime +7
-print
```

While this is a single command, you can see why you wouldn't want to have to type it in its entirety each time you wanted to find some files in your directory tree. To make our lives simpler, we'll make this command a shell script.

Turning your command list into a script

It's surprisingly easy to change a list of commands into a shell script. You need to do two things: You must enter the command list into a text file, and you must tell Solaris that the file is executable. While you could use an editor to create the script file, we'll use the technique described in the article "Quickly Create Small Text Files," on page 16.

```
cat > find.files
find $HOME -name '*.bak' -atime +7 -mtime +7 -print
[Ctrl] D
```

Now you've created a file named *find.files* that contains the list of commands you want to execute. (In this case, there's only one command, but you could have as many as you like in a single shell script.)

Next you must tell Solaris the file is executable using the `chmod` command. To do so, type

```
chmod a+x find.files
```

Now you have a shell script that can find all the old backup files from your directory tree. To run your macro, you just type the following line:

```
./find.files
```

You might be wondering why you need the ./ at the beginning of the line. That's because the directory you're currently in is presumably not on your path. When you type a command name at the shell prompt, Solaris searches through the directories stored in your path for the program to execute. The ./ tells Solaris to start looking in the current directory.

Please note: If you're going to be creating shell scripts of your own, you'll probably want to create a special directory for them. Then you can place this directory name in your path so you can run the shell scripts no matter what directory you're in.

You can do this by creating a directory off your home directory called *bin*. Then you can add your local *bin* directory to the path. In the Bourne and Korn shells, you can do so with the following commands:

```
cd $HOME
mkdir bin
PATH=$PATH:$HOME/bin
export PATH
```

If you're going to use the C shell, you can instead use

```
cd $HOME
mkdir bin
set path=($path $HOME/bin)
```

Now that you've created a local *bin* directory and placed it in your path, you can start placing shell scripts in your *bin* directory. Your shell scripts will start looking more like real Solaris commands because you no longer need to type the ./ before each macro name when you run it.

Please note that when you log out, Solaris will forget the changes you made to the path. You can fix this if you use the technique shown in the article "Execute Commands When Logging In" on page 15.

Command-line arguments

What do you do if there are other file types that you want to find? You could create a new shell script for each type of file. But what if you want to use a different aging period?

After you think about it a while, it's obvious that multiple shell scripts aren't the answer. Eventually you'd wind up with dozens of shell scripts, and you wouldn't be able to remember which one to use. Wouldn't it be nice to tell the shell script which types of files to delete and how old they should be?

It turns out that you can do this. Inside your shell script, you can refer to any arguments that you provide when you start it. When you put a **$** followed by a number into your script, the text the shell script sees will be the argument referenced by the number. For example, a **$3** will be replaced with the text of the third argument.

Two other sequences can be useful. The **$@** sequence represents a list of all the arguments, and **$#** represents the number of arguments passed to the shell script. As an example, let's assume you've created the following shell script named *test*:

```
echo You invoked test with $# arguments
echo Second: $2
echo First: $1
echo Entire list: $@
```

When you execute *test* with the command line

```
./test A private buffoon is a light-hearted
  loon
```

your shell script will print

```
You invoked test with 7 arguments
Second: private
First: A
Entire list: A private buffoon is a light-
  hearted loon
```

Arguments and quoting

When you invoke a macro and you use quotes in its arguments, you should be aware that anything inside a set of quotes is treated as a single argument, even if the quotes enclose one or more spaces. Thus, if we execute our test macro like this

```
./test "The first argument" "the second"
```

you'll see that *test* sees only two arguments.

As you probably know, when you use the * or ? characters to specify wildcards in filenames, the shell expands them for the program. So when you type

```
rm q*
```

the shell actually looks in the current directory, builds a list of all the filenames that start with a q, and passes this list to **rm**. What **rm** actually sees might be something like this:

```
rm quick quack quart
```

If you put the file specification in quotes, then the shell won't expand the file specification into the matching list of files. Thus, typing

```
rm 'q*'
```

tells the `rm` command to remove the file *q**. Similarly, when you write your own shell scripts, you need to be aware that if the user uses a wildcard specifier, the shell will expand the expression to match all possible files in the current directory and pass you the resulting list of arguments. If you don't want that, you need to enclose the argument in quotes.

Updating our shell script

Now let's add the ability for the script to accept some parameters that tell it how to behave. We'll use the first argument to describe the file extension *find* and the second argument to tell how old the files must be. To do so, all we have to do is modify our *find.files* script to look like this:

```
find $HOME -name '*.'$1 -atime +$2 -mtime +$2
-print
```

Please note that we put the `$1` outside the quotes for the file specification rather than inside the quotes. If we leave the `$1` inside the quotes, the script will look for all files with the ending `.$1` rather than what we want. We'll cover the rules on how to determine where to put quotes and such next month when we discuss variables in shell scripts.

Our shell script is now much more flexible. Rather than only finding *.bak* files that haven't been used in a week, you can use it to find files with any given extension that haven't been used for as many days as you specify.

Comments

Even though the shell scripts in these examples aren't very complex, you'll want to adopt the habit of documenting your shell scripts. As you get better at writing shell scripts, you'll find that your scripts get more complex. As you maintain them, you'll find that commenting your shell scripts will help you to modify them for new uses with a minimum of hassle.

If you start a line in your shell script with the # symbol, the entire line is ignored. Therefore, you can use this feature to add documentation to your shell scripts. At a minimum, you should describe the purpose of your shell script. In the places where your script gets complex, you'll want to add comments to help your reader decipher what the script is doing. Figure A shows our shell script with some comments added.

Figure A

```
# find.files arg1 arg2
# This script file finds all files ending with the
# extension in arg1 that haven't been accessed or
# modified in the last arg2 days
#
# Note: The *. is in quotes to prevent the shell from
# replacing it with a list of file names found in the
# current directory. The $1 is outside of the quotes
# so that the shell *will* replace it.
#.
find $HOME -name '*.'$1 -atime +$2 -mtime +$2 -print
```

This shell script can help you locate unused files on your hard drive.

Conclusion

Writing shell scripts can simplify your life by making many tasks easier. Rather than memorizing complex sequences of instructions, you can make your own shorthand. As the example we just used shows, creating a shell script doesn't have to be particularly difficult. ❖

quick tip

Create multiple directories with mkdir

Have you ever needed to create a directory tree like */test/app/data*, */test/app/bin*, and */test/app/config* when you haven't yet even created the */test* directory? You might think that you need to execute five commands to do so:

```
mkdir /test
mkdir /test/app
mkdir /test/app/data
mkdir /test/app/bin
mkdir /test/app/config
```

It turns out that you can do the same job with only three commands instead of five. The `mkdir` command has a switch, `-p`, that tells `mkdir` to create any parent directory that doesn't already exist. This allows you to create multiple directories with a single command. Thus, if you type the command

```
mkdir -p /test/app/data
```

the `mkdir` command first finds that the */test* directory doesn't exist, so it creates it. Then it notices that the */test/app* directory doesn't exist and creates it as well. Finally, it creates the *data* subdirectory. You can then finish the job by issuing the commands:

```
mkdir /test/app/bin
mkdir /test/app/config
```

The next time you need to create a complex directory tree, save yourself some typing and try the `-p` option on the `mkdir` command. ❖

letters

Adding external disk causes system failure

Our system has been very reliable in the past. However, we added an external disk drive last week, and now it crashes at least once a day. Can you help?

William Scotts
Dayton, Ohio

We normally don't print letters like this, but after some work on the phone, we've discovered that this is a classic problem that people should be reminded of. It turns out that William's problem was an improperly terminated SCSI cable.

William was using an Adaptec 1542 SCSI controller, which allows both internal and external drives. Originally, he had two internal drives, a CD-ROM and a 1GB hard drive. His system was cabled, as shown in Figure A. As you can see, the 1GB hard drive has a terminator on it.

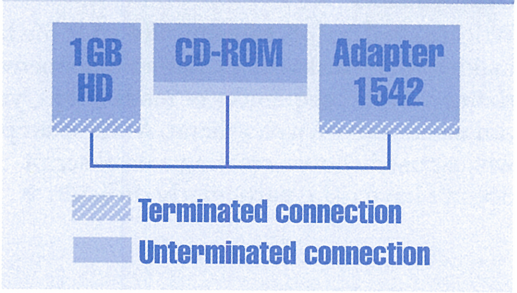
The rules of a SCSI interface say that *both* ends of a SCSI cable must be terminated. The Adaptec 1542 has terminators on it by default. Thus the 1GB hard drive at one end of the cable is terminated, and the other end of the cable is terminated on the card.

For the Adaptec 1542, like many other cards, the external connector is an extension of the internal connector. When the external hard drive was added to the external connector, the SCSI cable had terminators on both ends *and* in the middle, as shown in Figure B. This caused the system crashes William is experiencing. In this case, all that was required was to remove the terminators on the Adaptec 1542 card.

Please note that some SCSI controller cards might have the external connector on a different SCSI bus. In this case, the internal and external connectors are on different cables. On such drives, you may not have to remove a terminator from the controller card.

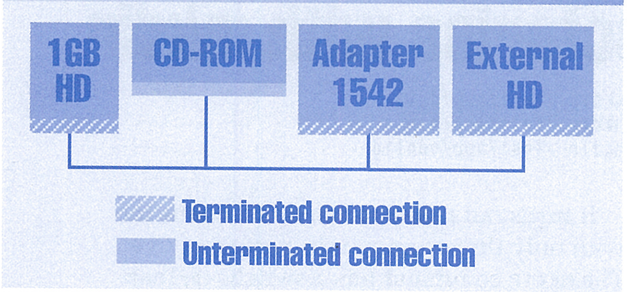
Remember, at the physical end of each SCSI bus, you need to have a terminator. Read the documentation on your controller card to find out if the internal and external connectors are on the same SCSI bus and to determine the procedure to mix internal and external SCSI devices.

Figure A



The system configuration appeared like this before the external hard drive was installed.

Figure B



Since a SCSI bus must be terminated only at the ends, we should remove the terminators on the SCSI adapter.

Connecting to Cobb's anonymous FTP server

While I was reading the February issue of *Inside Solaris*, I ran across the article on your anonymous FTP site (*"Inside Solaris* List Server and Anonymous FTP Update"). I tried to connect to the site with the following command

```
ftp ftp.cobb.zd.com/pub/COBB/solaris
```

but was rejected, like so

```
ftp.cobb.zd.com/pub/COBB/solaris: unknown host
```

What's wrong?

Terry Holthaus
New York City

When we published the address of the FTP site, we used the popular convention of appending the directory path to the machine name to show you where the files are. Actually, you have to specify the machine name when you start `ftp`. Then after you've established a connection, you use the `cd` command to go to the proper directory. The following dialog will show you how to do this:

```
ftp ftp.cobb.zd.com
Connected to ias3.iacnet.com.
220 ias3 FTP server (Version wu-2.4(3)
   Wed May 24 21:37:02 EDT 1995) ready.
User (ias3.iacnet.com:(none)): anonymous
331 Guest login ok, send your complete
   e-mail address as password.
Password:
230 Guest login ok, access restrictions apply.
ftp> cd pub/COBB/solaris
250 CWD command successful.
ftp>
```

As a general rule, when you see an FTP address with forward slashes in it like *ftp.cobb.zd.com/pub/COBB/solaris*, just use the text before the first slash as the machine name for establishing your FTP connection. Then use the text after the first slash for the directory name. Now you should be able to log in and use our FTP site without any problems. ❖

using shells effectively

Execute commands when logging in

Have you ever needed to set the value of an environment variable to make some application run correctly? If so, you've probably noticed that when you log in the next time, you have to reset the environment variable to run your program again. Perhaps after you log in, you always run the same couple of commands, just to see how the system is doing.

If you're in one of these situations, then you'll benefit from this discussion. In this article, we're going to discuss how to make your shell automatically execute commands for you when you log in to your computer.

Which shell are you running?

Before you try using the procedures we're about to present, you need to know which shell you're running when you log in. To find this out, execute the following command:

```
cat /etc/passwd | grep \^user_name
```

Be sure to replace *user_name* with your user name. This command will scan the */etc/passwd* file and find the entry for your name. Figure A shows the results of searching for the user marco on one of our systems.

Figure A

```
$ cat /etc/passwd | grep \^marco
marco:x:1313:60001:Editor-In-Chief, Inside
Solaris:/export/home/marco:/bin/sh
$
```

Here we found the /etc/passwd entry for the user marco.

Notice that the second line in Figure A contains many fields of information separated by colons. The first field holds the user name, which is the key we used in our search. The last field holds the path of the shell to execute. In this case, the default shell is the Bourne shell.

The three shells shipped with Solaris are the Bourne shell, the C shell, and the Korn shell. By default, when you create a new

SunSoft Technical Support
(800) 786-7638

SECOND CLASS MAIL

5147831 02/97
[redacted]

Please include account number from label with any correspondence.

user account, the account uses the Bourne shell unless the system administrator overrides it. The files */bin/sh* and *bin/jsh** hold the code for the Bourne shell, the file */bin/csh* holds the code for the C shell, and the files */bin/ksh* and */bin/rksh** hold the code for the Korn shell. (* These are special versions of the shell.)

The Bourne and Korn shells

When you log in using the Bourne or Korn shell, the shell automatically looks for two shell scripts to execute for you. First, the shell looks for the script */etc/profile* and executes it if it exists. You should place any commands you want executed by all users in this file. Next, the shell looks in your home directory for a script named *.profile*, executing it if found. Each user uses the *.profile* file to customize his or her environment.

The Korn shell performs an additional step: It looks for the environment variable ENV and searches for the file it specifies. If it finds one, then it executes that script as well.

The C shell

When you log in using the C shell, the C shell looks in your home directory for a file named *.cshrc* and executes it if found. You use this script file to customize your environment.

If you've just logged in to the computer, the C shell executes the *.login* script file found in your home directory, if it exists. You use this script file to run any commands that you need to run only once per login, such as setting your path or customizing your terminal settings. Please note that the C shell executes the *.login* script after the *.cshrc* script, if it executes the *.login* script at all.

The C shell provides one additional feature you might want to use: It lets you execute a script file when you log out of the computer. To do so, simply create a script named *.logout*, and when you log out, the C shell will execute it. (Please note that this technique works only when you're using the C shell as your login shell.) ❖

quick tip

Quickly create small text files

If you're like most of us, you often want to create a small text file. While you can run `vi` or start the text editor to do so, there's a quicker way for tiny files. You can use the `cat` command to redirect the characters you type on the keyboard to a file.

To do so, just type *cat >fname*, where *fname* is the name of the file you want to create. When you press [Enter], Solaris will put the cursor on a blank line. Just type the text you want to store in your file, and press [Ctrl]D when you're finished. For example, if you want to put the sentence "I am the very model of a modern Major-General" into a text file named *Pirates.ditty*, you can simply type this:

```
cat >Pirates.ditty
I am the very model of a modern Major-
   General
[Ctrl] D
```

This technique can save you some time since you don't have to load a big editor like `vi` from the disk to quickly create small text files. If you're all thumbs, just remember to correct your errors on each line before pressing [Enter]. If you don't, the file will contain errors, and you'll either have to redo it or use `vi` anyway. ❖

Printed in the USA
This journal is printed on recyclable paper.